Shobhit
Institute of Engineering & Technology
Deemed to-be-University
EDUCATION EMPOWERS

COMPILER DESIGN

Prepared By:
AVINAV PATHAK
Assistant Professor
Shobhit Institute of Engineering and Technology
(Deemed to-be- University) MEERUT - 250110

Shobhit
Institute of Engineering & Technology
Deemed to-be-University
EDUCATION EMPOWERS

CONTENTS

1.Compilers and Translators : Introduction

2. Lexical Analysis

3. Implementation of Lexical Analyzer

4. Basic Parsing Techniques

5. Syntax Analyzer Generator

6. Run Time Memory Management

7. Error Detection and Recovery

8. Code Optimization and Code Generation

What is a Compiler?

Introduction to Compiler

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- 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.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- 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.

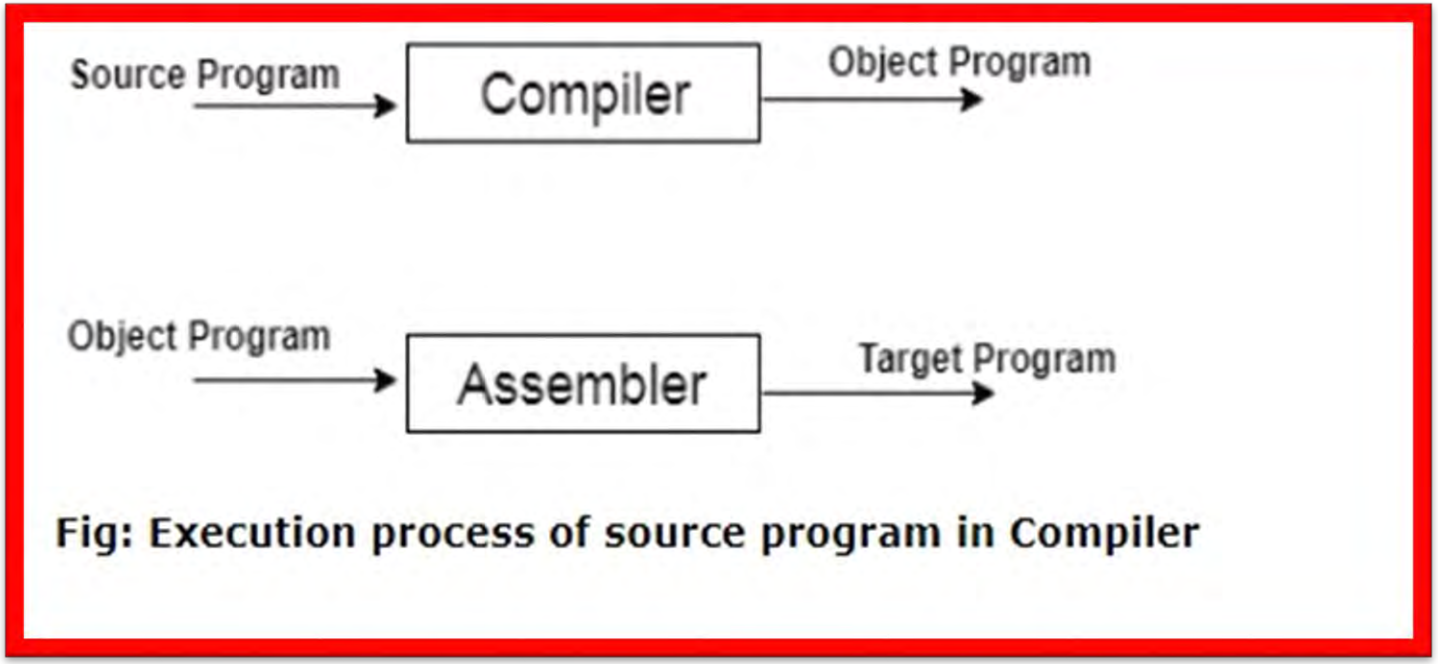


Fig: Execution process of source program in Compiler

Phases of Compiler

Compiler Phases

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:

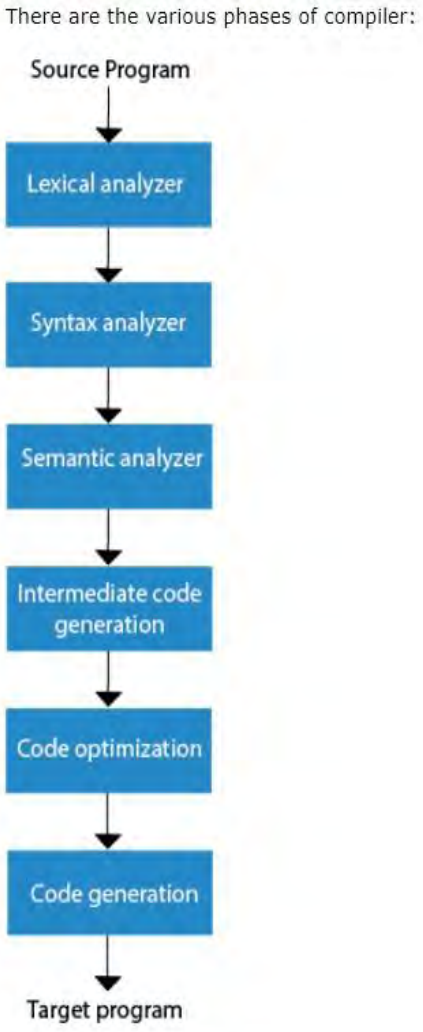


Fig: phases of compiler

Phases of Compiler

Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

Syntax Analysis

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

Semantic Analysis

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

Phases of Compiler

Intermediate Code Generation

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

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Pass Structure Compiler

Compiler Passes

Pass is a complete traversal of the source program. Compiler has two passes to traverse the source program.

Multi-pass Compiler

Multi pass compiler is used to process the source code of a program several times.
In the first pass, compiler can read the source program, scan it, extract the tokens and store the result in an output file.

In the second pass, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.

In the third pass, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax.

This pass is going on, until the target output is produced.

One-pass Compiler

One-pass compiler is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
In the one pass compiler, when the line source is processed, it is scanned and the token is extracted.

Then the syntax of each line is analyzed and the tree structure is build. After the semantic part, the code is generated.

The same process is repeated for each line of code until the entire program is compiled.

Language Processing System

Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.

The C compiler, compiles the program and translates it to assembly program (low-level language).

An assembler then translates the assembly program into machine code (object).

A linker tool is used to link all the parts of the program together for execution (executable machine code).

A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

Assembler

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

Linker

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

Loader

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

Cross-compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

ASSIGNMENT

Questions:

1. Explain Pass Structure of Compiler.
2. Discuss phases of Compiler.
3. Discuss Language Processing System

Bootstrapping

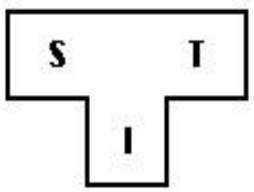
Bootstrapping is widely used in the compilation development.

Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

A compiler can be characterized by three languages:

Source Language
Target Language
Implementation Language

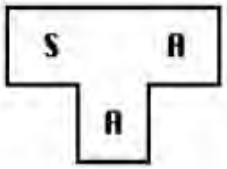
The T- diagram shows a compiler SCIT for Source S, Target T, implemented in I.



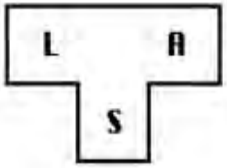
Follow some steps to produce a new language L for machine A:

Bootstrapping

1. Create a compiler ${}^{S}C_{A}{}^{A}$ for subset, S of the desired language, L using language "A" and that compiler runs on machine A.

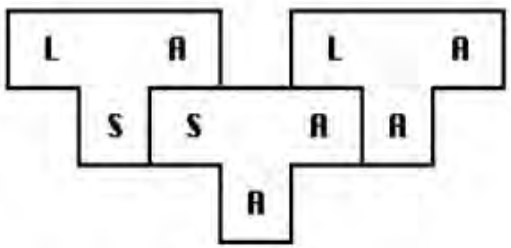


2. Create a compiler ${}^{L}C_{S}{}^{A}$ for language L written in a subset of L.



3. Compile ${}^{L}C_{S}{}^{A}$ using the compiler ${}^{S}C_{A}{}^{A}$ to obtain ${}^{L}C_{A}{}^{A}$. ${}^{L}C_{A}{}^{A}$ is a compiler for language L, which runs on machine A and produces code for machine A.

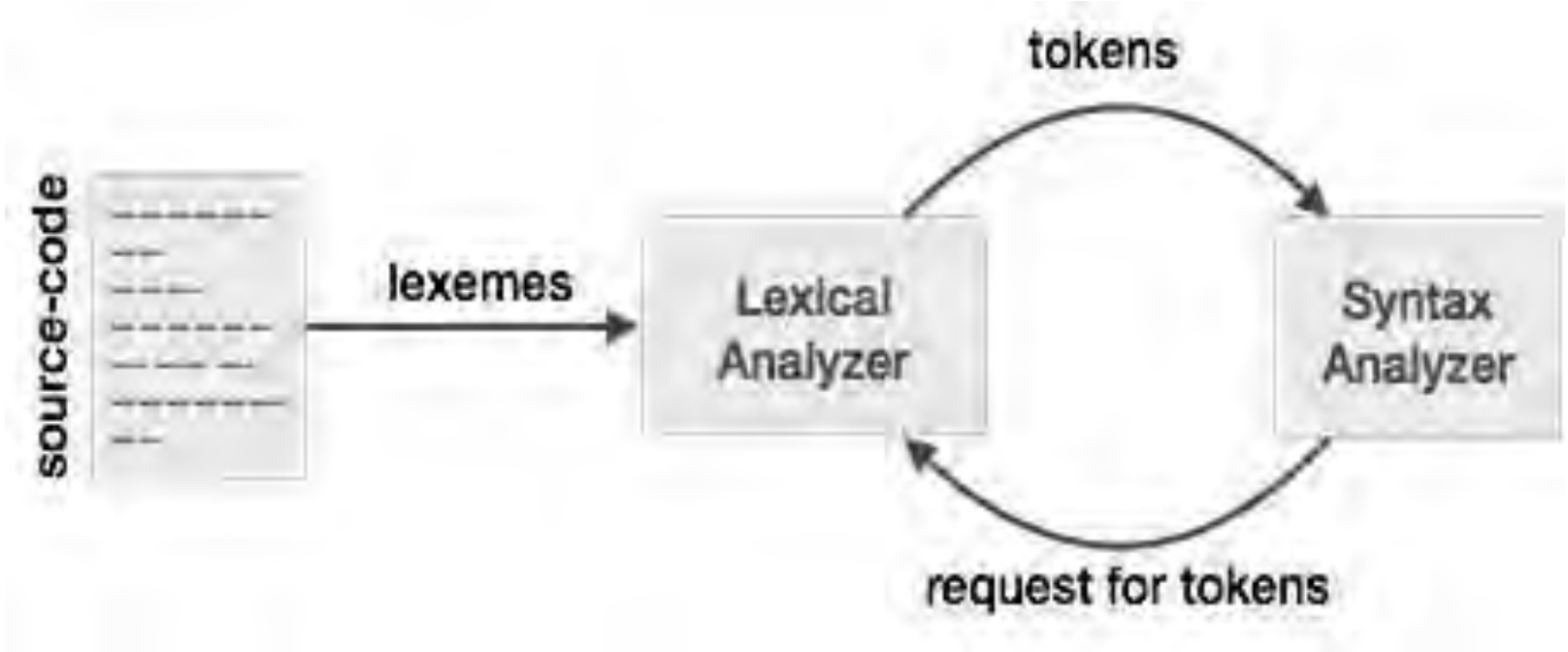
$$ {}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A} $$



Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Lexical Analysis

Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.
For example, in C language, the variable declaration line
int value = 100;
contains the tokens:
int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).
Specifications of Tokens
Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by **ε** (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Lexical Analysis

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

ASSIGNMENT

Questions:

1. Explain phases of Lexical Analysis in detail.
2. What is Lexical Analysis?
3. What is Bootstrapping? Discuss in detail

Regular Expressions

Regular Expression

❖Regular expression is a sequence of pattern that defines a string. It is used to denote regular languages.
❖It is also used to match character combinations in strings. String searching algorithm used this pattern to find the operations on string.
❖In regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx,.....}
❖In regular expression, x+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx,.....}

Operations on Regular Language

The various operations on regular language are:

1. Union: If L and M are two regular languages then their union L U M is also a union. L U M = {s | s is in L or s is in M}
2. Intersection: If L and M are two regular languages then their intersection is also an intersection. L ∩ M = {st | s is in L and t is in M}
3. Kleene closure: If L is a regular language then its kleene closure L1* will also be a regular language. L* = Zero or more occurrence of language L.

Regular Expressions

Example

Write the regular expression for the language:

L = {abn w:n ≥ 3, w ∈ (a,b)+}

Solution:
The string of language L starts with "a" followed by atleast three b's. Itcontains atleast one "a" or one "b" that is string are like abbba, abbbbbba, abbbbbbbb, abbbb.....a

So regular expression is:

r= ab3b* (a+b)+
Here + is a positive closure i.e. (a+b)+ = (a+b)* - ∈

Finite State Machines and Transition Diagrams

Finite state machine

Finite state machine is used to recognize patterns.
Finite automata machine takes the string of symbol as input and changes its state accordingly. In the input, when a desired symbol is found then the transition occurs.
While transition, the automata can either move to the next state or stay in the same state.

FA has two states: accept state or reject state. When the input string is successfully processed and the automata reached its final state then it will accept.

A finite automata consists of following:
Q: finite set of states
∑: finite set of input symbol
q0: initial state
F: final state
δ: Transition function
Transition function can be define as
δ: Q x ∑ →**Q**
FA is characterized into two ways:

DFA (finite automata)
NDFA (non deterministic finite automata)

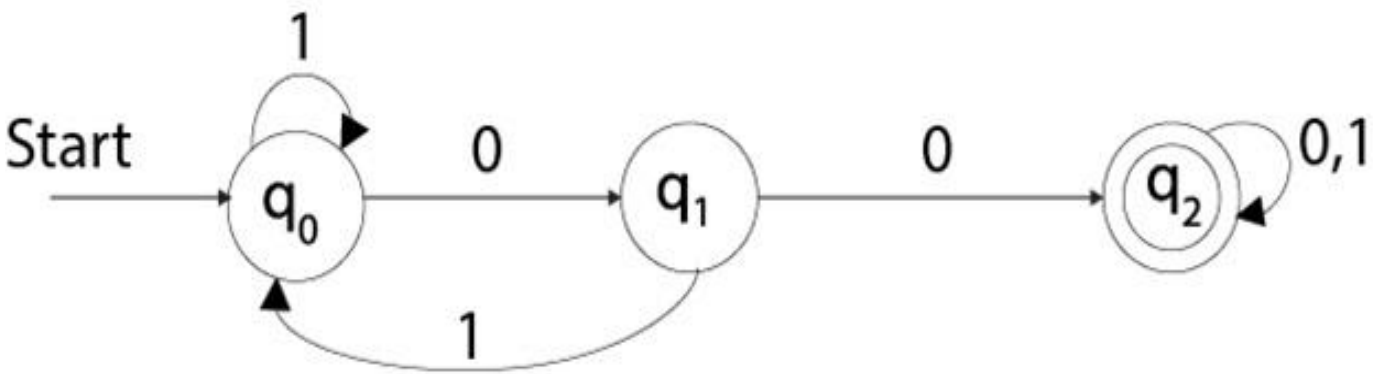
Finite State Machines and Transition Diagrams

DFA has five tuples {Q, ∑, q0, F, **δ**}

Q: set of all states
∑: finite set of input symbol where **δ**: Q x ∑ →**Q**
q0: initial state
F: final state
δ: Transition function

Example
See an example of deterministic finite automata:

Q = {q0, q1, q2}
∑ = {0, 1}
q0 = {q0}
F = {q3}
Finite state machine



Finite State Machines and Transition Diagrams

NDFA

NDFA refer to the Non Deterministic Finite Automata. It is used to transit the any number of states for a particular input. NDFA accepts the NULL move that means it can change state without reading the symbols.

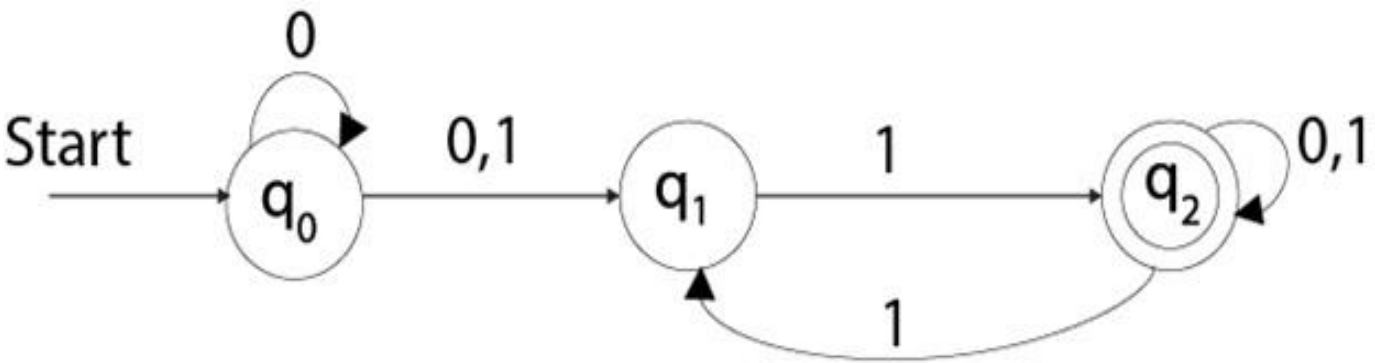
NDFA also has five states same as DFA. But NDFA has different transition function.

Transition function of NDFA can be defined as:
δ: Q x ∑ →2Q

Example

See an example of non deterministic finite automata:

Q = {q0, q1, q2}
∑ = {0, 1}
q0 = {q0}
F = {q3}



ASSIGNMENT

Questions:

1. Discuss Transition diagrams in Finite Automata.
2. What are Regular Expressions? Explain using and example.

Parsing

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 paring
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

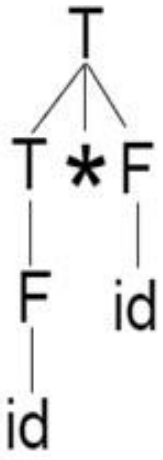
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow id$
$F \rightarrow T$
$F \rightarrow id$

Parsing

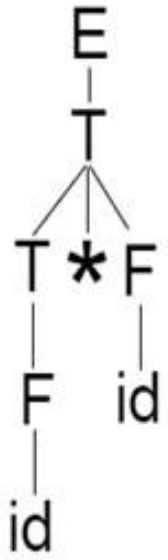
Bottom up parsing is classified in to various parsing. These are as follows:

Shift-Reduce Parsing
Operator Precedence Parsing
Table Driven LR Parsing
LR(1)
SLR(1)
CLR (1)
LALR(1)

Step 4



Step 5

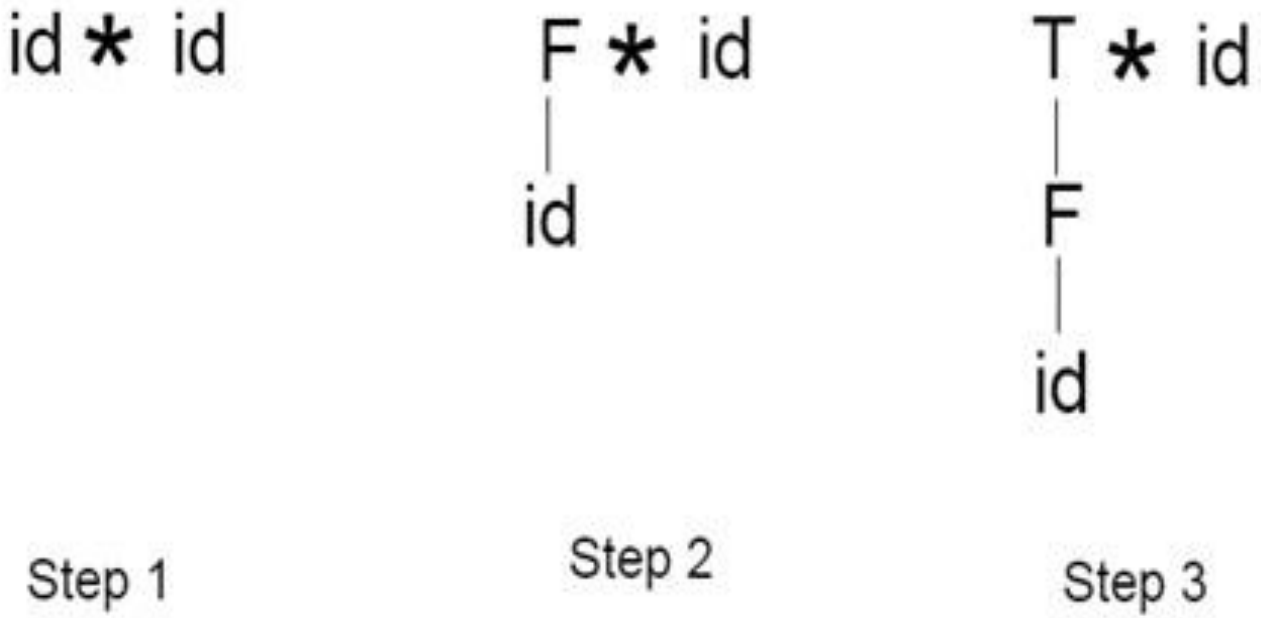


Step 6

Parsing

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

Shift-Reduce Parsing
Operator Precedence Parsing
Table Driven LR Parsing
LR(1)
SLR(1)
CLR (1)
LALR(1)



Shift Reduce Parsing

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.

Shift reduce parsing

Sift 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 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 → S+S
S → S-S
S → (S)
S → a

Shift Reduce Parsing

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

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∈.
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 ≐ b means that the terminal "a" and "b" both have same precedence.

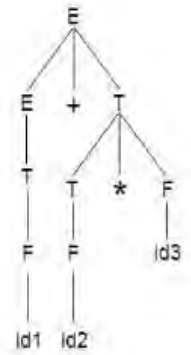
Operator Precedence Parsing

Example
Grammar:

E → E+T/T
T → T*F/F
F → id
Given string:

w = id + id * id
Let us consider a parse tree for it as follows

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	≐	X	⋗
T	X	X	X	X	⋗	≐	⋗
F	X	X	X	X	⋗	⋗	⋗
id	X	X	X	X	⋗	⋗	⋗
+	X	=	⋖	⋖	X	X	X
*	X	X	=	⋖	X	X	X
$	⋖	⋖	⋖	⋖	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 ≐ + ⋖ id2 ⋗ * id3 $

$ ⋖ E ≐ + ⋖ F ⋗ * id3 $

$ ⋖ E ≐ + ⋖ T ≐ * ⋖ id3 ⋗ $

$ ⋖ E ≐ + ⋖ T ≐ * ≐ F ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ⋗ $

Accept.

Operator Precedence Parsing

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.

ASSIGNMENT

Questions:

1. Discuss Shift reduce Parsing Algorithm using an example.
2. What is Operator Precedence parsing? Discuss.

LALR(1) Parsing

LALR (1) Parsing:

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items
LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.
Example
LALR (1) Grammar
S → AA
A → aA
A → b
Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b
I0 State:

Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure (S` → •S)

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

I0 = S` → •S, $
S → •AA, $

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

LALR(1) Parsing

I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)

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

I2= S → A•A, $
A → •aA, $
A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
I6= Go to (I2, a) = Closure (A → a•A, $)

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

LALR(1) Parsing

I6 = A → a•A, $
A → •aA, $
A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { A → a•A, a/b
A → •aA, a/b
A → •b, a/b
}

I6= { A → a•A, $
A → •aA, $
A → •b, $
}

LALR(1) Parsing

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/$
A → •aA, a/b/$
A → •b, a/b/$
}

The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, a/b/$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/$}

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:

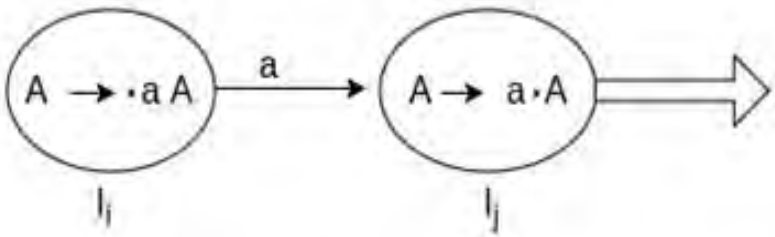
If a state (Ii) is going to some other state (Ij) on a terminal then it corresponds to a shift move in the action part.

SLR(1) Parsing

SLR (1) Table Construction

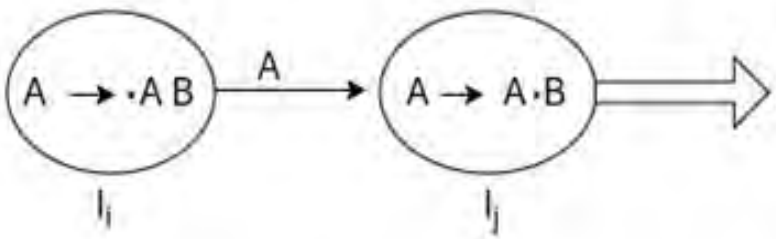
The steps which use to construct SLR (1) Table is given below:

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.



States	Action		Go to
	a	$	A
I_i I_j	Sj		

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.



States	Action		Go to
	a	$	A
I_i I_j			j

SLR(1) Parsing

Example
S -> •Aa
A->**αβ**•
Follow(S) = {$}
Follow (A) = {a}

SLR (1) Grammar

S → E
E → E + T | T
T → T * F | F
F → id

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

S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •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` → •E
E → •E + T
E → •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.

SLR(1) Parsing

Example
S -> •Aa
A->**αβ**•
Follow(S) = {$}
Follow (A) = {a}

SLR (1) Grammar

S → E
E → E + T | T
T → T * F | F
F → id

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

S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I0 State:

Add Augment production to the I0 State and Compute the Closure

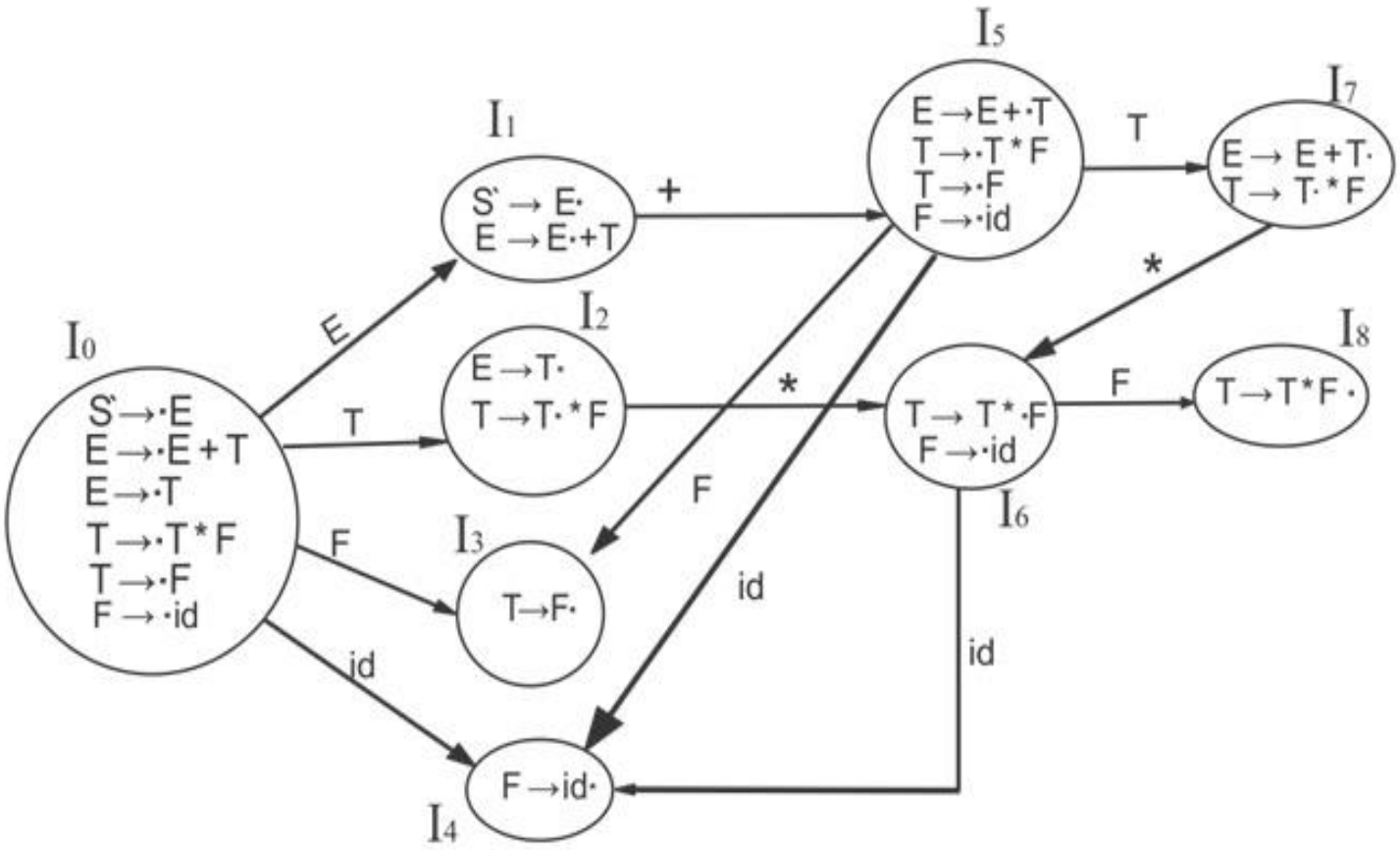
I0 = Closure (S` → •E)

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

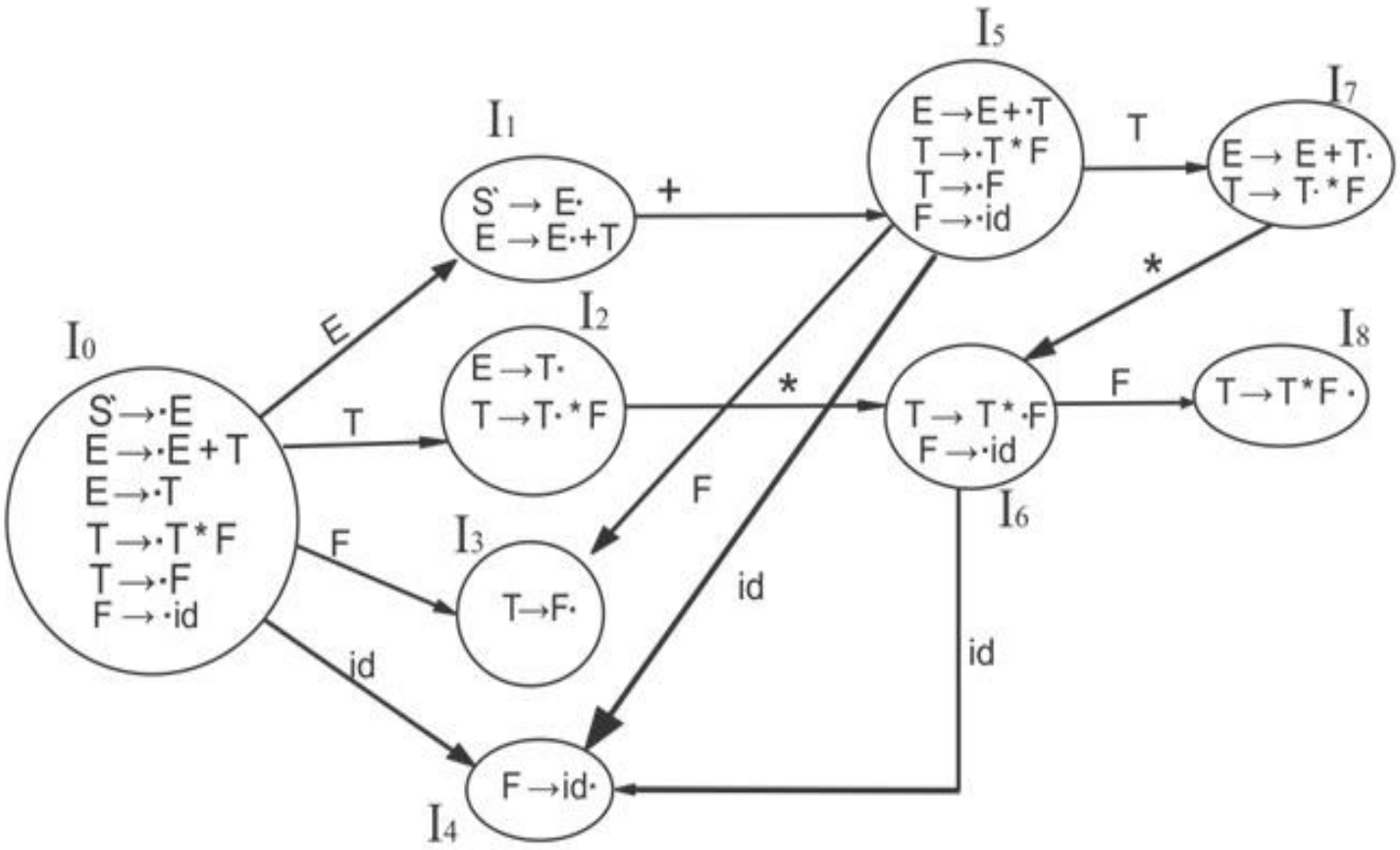
I0 = S` → •E
E → •E + T
E → •T

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

SLR(1) Parsing



SLR(1) Parsing



CLR(1) Parsing

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (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 CLR (1) parsing table
LR (1) item

LR (1) item is a collection of LR (0) items and a look ahead symbol.

LR (1) item = LR (0) item + look ahead

The look ahead is used to determine that where we place the final item.

The look ahead always add $ symbol for the argument production.

CLR(1) Parsing

Example
CLR (1) Grammar

S → AA
A → aA
A → b
Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b
I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

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

I0 = S` → •S, $
S → •AA, $

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

I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)

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

I2= S → A•A, $
A → •aA, $
A → •b, $

CLR(1) Parsing

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
I6= Go to (I2, a) = Closure (A → a•A, $)

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

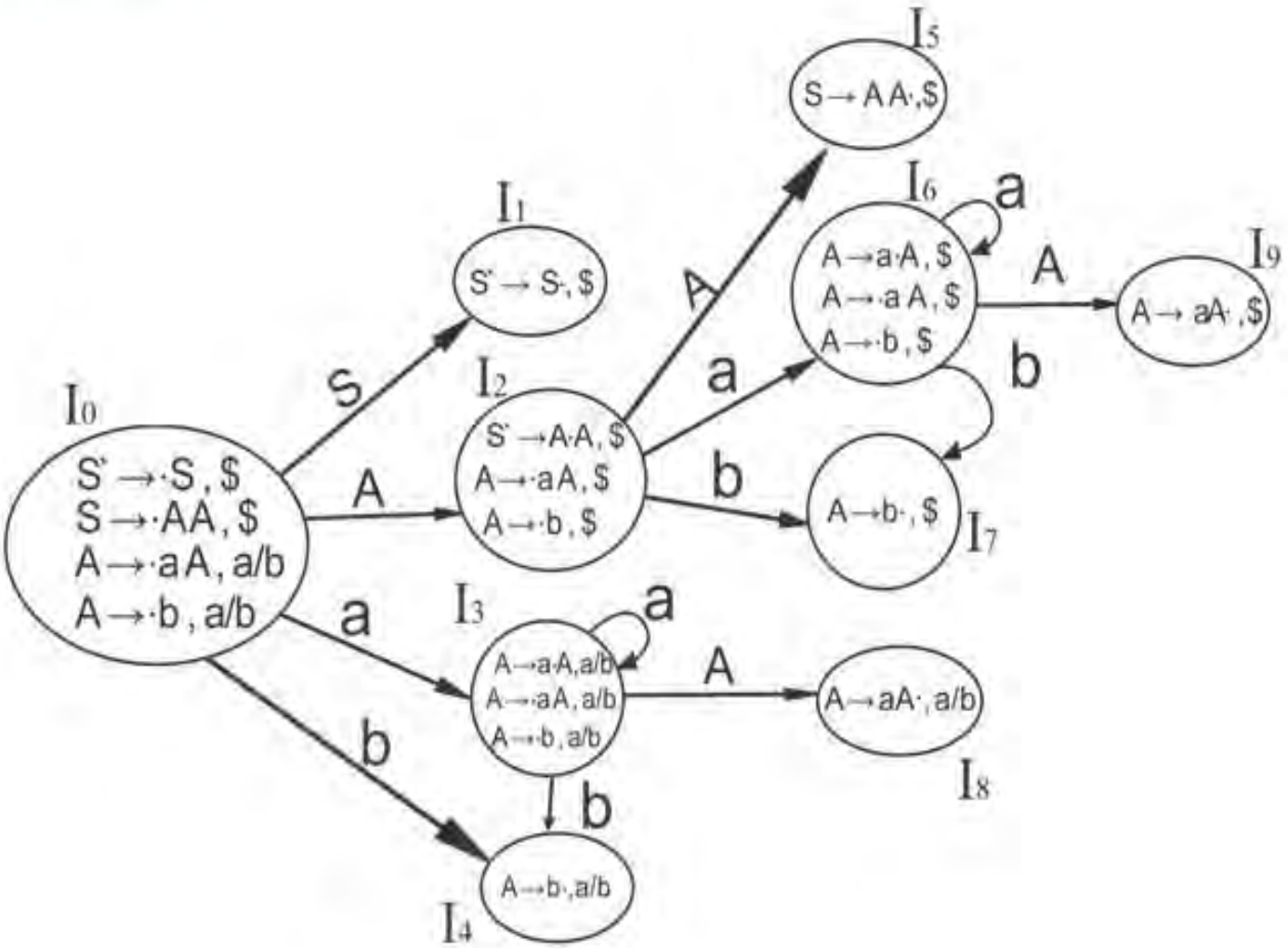
I6 = A → a•A, $
A → •aA, $
A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

CLR(1) Parsing

Drawing DFA:



ASSIGNMENT

Questions:

1. Write short notes on : (i) SLR(1) and (ii) LR(1) Parsing
2. Distinguish between CLR(1) and LALR(1) Parsing?

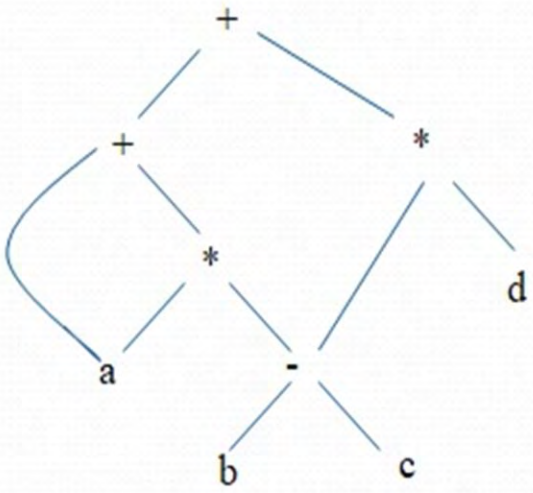
Three Address Code

- Intermediate code is the interface between front end and back end in a compiler

- Ideally the details of source language are confined to the front end and the details of target machines to the back end (a m*n model)

- In this chapter we study intermediate representations, static type checking and intermediate code generation

In a three address code there is at most one operator at the right side of an instruction



t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

FORMS OF 3 ADDRESS INSTR

- x = y op z
- x = op y
- x = y
- goto L
- if x goto L and ifFalse x goto L
- if x relop y goto L
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- x = y[i] and x[i] = y
- x = &y and x = *y and *x =y

EXAMPLE

- do i = i+1; while (a[i] < v);

L:	t1 = i + 1	100:	t1 = i + 1
	i = t1	101:	i = t1
	t2 = i * 8	102:	t2 = i * 8
	t3 = a[t2]	103:	t3 = a[t2]
	if t3 < v goto L	104:	if t3 < v goto 100
	Symbolic labels		Position numbers

DATA STRUCTURES FOR THREE ADDRESS CODES

Quadruples

- Has four fields: op, arg1, arg2 and result

Triples

- Temporaries are not used and instead references to instructions are made

Indirect triples

- In addition to triples we use a list of pointers to triples

EXAMPLE

- b * minus c + b * minus c

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

Quadruples

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect Triples

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Introduction

- 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
$F \rightarrow num$	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

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 → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Implementation of Syntax directed translation

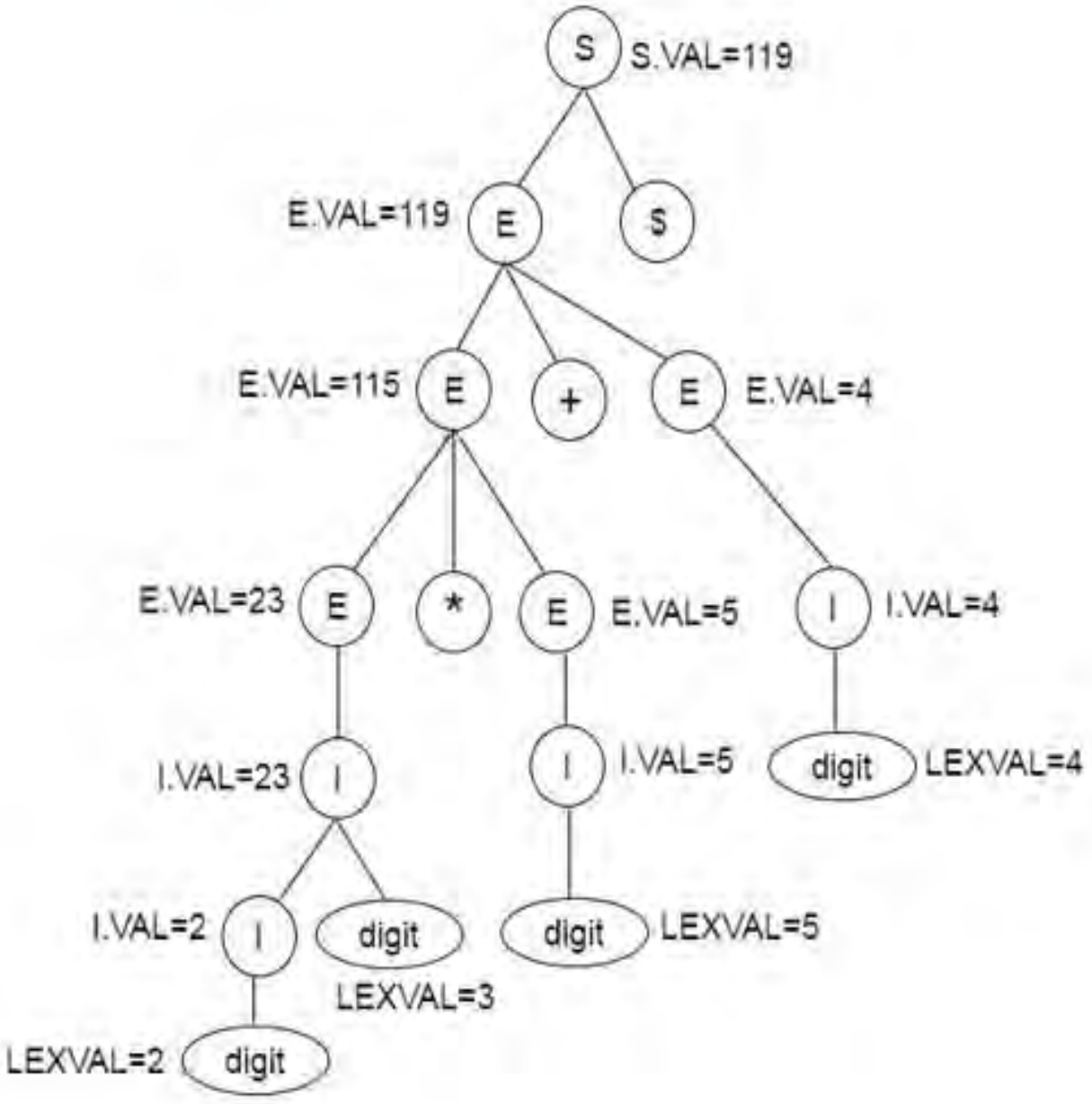
- Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

- SDT is implementing by parse the input and produce a parse tree as a result.

Example

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

Parse tree for SDT:



Introduction

- 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

$$
\begin{aligned}
S &\rightarrow id := E \\
E &\rightarrow E1 + E2 \\
E &\rightarrow E1 * E2 \\
E &\rightarrow (E1) \\
E &\rightarrow id
\end{aligned}
$$

The translation scheme of above grammar is given below:

Production rule	Semantic actions

S → id :=E	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

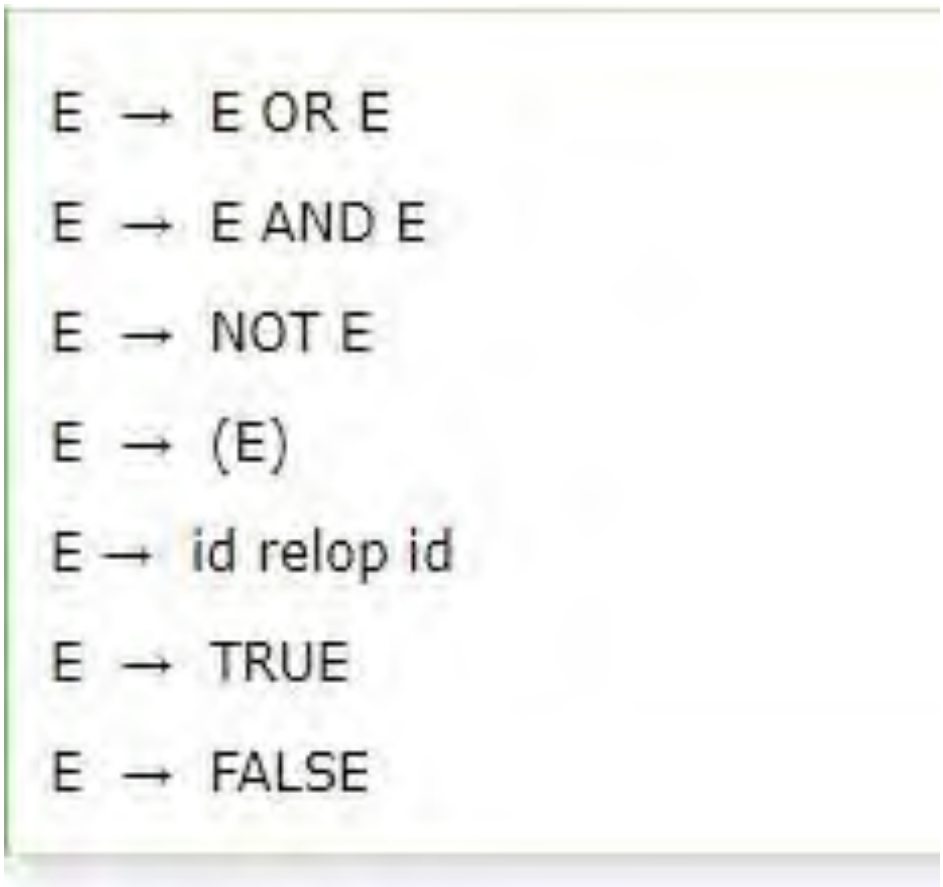
E → E1 + E2	{E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }
E → (E1)	{E.place = E1.place}

E → id	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

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



The relop is denoted by <, >, <, >.

Production rule	Semantic actions
E → E1 OR E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3); EMIT (E.place ':=' '0') EMIT ('goto' nextstat + 2) EMIT (E.place ':=' '1') }
E → TRUE	{E.place := newtemp(); Emit (E.place ':=' '1') }
E → FALSE	{E.place := newtemp(); Emit (E.place ':=' '0') }

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

TRANSLATION OF FLOW CONTROL STATEMENTS

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 → LABEL : S

LABEL → 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 → **if** E then S

S → **if** E then S **else** S

S → **while** E **do** S

S → begin L end

S→ A

L→ L ; S

L → S

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 → **if** E then M S

S → **if** E then M S **else** M S

S → **while** M E **do** M S

S → begin L end

S → A

L → L ; M S

L → S

M → ϵ

N → ϵ

The translation scheme for this grammar is as follows:

Production rule	Semantic actions
S → if E then M S1	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
S → if E then M1 S1 else M2 S2	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
S → while M1 E do M2 S1	BACKPATCH (S1,NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
S → begin L end	S.NEXT = L.NEXT
S → A	S.NEXT = MAKELIST ()
L → L ; M S	BACKPATHCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
L → S	L.NEXT = S.NEXT
M → ϵ	M.QUAD = NEXTQUAD
N→ ϵ	N.NEXT = MAKELIST (NEXTQUAD) GEN (goto_)

ARRAY REFERENCES IN ARITHMETIC EXPRESSIONS

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive location. Array can be one dimensional or two dimensional.

For one dimensional array:

A: array[low..high] of the ith elements is at:
base + (i-**low)*width** → i*width + (base - low*width)
Multi-dimensional arrays:
Row major or column major forms

Row major: a[1,1], a[1,2], a[1,3], a[2,1], a[2,2], a[2,3]
Column major: a[1,1], a[2,1], a[1, 2], a[2, 2],a[1, 3],a[2,3]

In raw major form, the address of a[i1, i2] is
Base+((i1-low1)*(high2-low2+1)+i2-low2)*width
Translation scheme for array elements
Limit(array, j) returns nj=highj-lowj+1

place: the temporary or variables

offset: offset from the base, null if not an array reference

The production:

$S \rightarrow L := E$

$E \rightarrow E+E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist\ ]$

$L \rightarrow id$

$Elist \rightarrow Elist, E$

$Elist \rightarrow id[E$

Production Rule	Semantic Action

S → L := E	{if L.offset = null then emit(L.place ':=' E.place)
	else EMIT (L.place'['L.offset ']' ':=' E.place);
	}

E → E+E	{E.place := newtemp; EMIT (E.place ':=' E1.place '+' E2.place); }
E → (E)	{E.place := E1.place;}

E → L	{if L.offset = null then E.place = L.place
	else {E.place = newtemp;
	EMIT (E.place ':=' L.place '[' L.offset ']';
	}
	}

L → Elist]	{L.place = newtemp; L.offset = newtemp; EMIT (L.place ':=' c(Elist.array)); EMIT (L.offset ':=' Elist.place '*' width(Elist.array); }
L → id	{L.place = lookup(id.name); L.offset = null; }

Elist → Elist, E	{t := newtemp;
	m := Elist1.ndim + 1;
	EMIT (t ':=' Elist1.place '*' limit(Elist1.array, m));
	EMIT (t, ':=' t '+' E.place);
	Elist.array = Elist1.array;
	Elist.place := t;
	Elist.ndim := m;
	}

Elist → id[E	{Elist.array := lookup(id.name); Elist.place := E.place Elist.ndim := 1; }

Postfix Translation

- In a production A → **α,** 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 → **while** M1 E **do** M2 S1

Can be factored as:

S → C S1
C → W E **do**
W → **while**

A suitable transition scheme would be

Production Rule	Semantic Action
W → while	W.QUAD = NEXTQUAD
C → W E do	C W E do
S→ C S1	BACKPATCH (S1.NEXT, C.QUAD) S.NEXT = C.FALSE GEN (goto C.QUAD)

Postfix translation of for statement

The production

S for L = E1 step E2 to E3 do S1

Can be factored as

F → for L

T → F = E1 by E2 to E3 do

S → T S1

Array References Translation

•Array references in arithmetic expressions

•Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive location. Array can be one dimensional or two dimensional.

•For one dimensional array:

•A: array[low..high] of the ith elements is at:
•base + (i-**low)*width** → i*width + (base - low*width)
•Multi-dimensional arrays:
•Row major or column major forms

•Row major: a[1,1], a[1,2], a[1,3], a[2,1], a[2,2], a[2,3]
•Column major: a[1,1], a[2,1], a[1, 2], a[2, 2],a[1, 3],a[2,3]
•In row major form, the address of a[i1, i2] is
•Base+((i1-low1)*(high2-low2+1)+i2-low2)*width
•Translation scheme for array elements
•Limit(array, j) returns nj=highj-lowj+1

$S \rightarrow L := E$

$E \rightarrow E+E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist\ ]$

$L \rightarrow id$

$Elist \rightarrow Elist, E$

$Elist \rightarrow id[E$

A suitable transition scheme for array elements would be:

A suitable transition scheme for array elements would be:

Production Rule	Semantic Action

S → L := E	{if L.offset = null then emit(L.place ':=' E.place)
	else EMIT (L.place'['L.offset ']' ':=' E.place);
	}

E → E+E	{E.place := newtemp; EMIT (E.place ':=' E1.place '+' E2.place); }
E → (E)	{E.place := E1.place;}

E → L	{if L.offset = null then E.place = L.place
	else {E.place = newtemp;
	EMIT (E.place ':=' L.place '[' L.offset ']');
	}
	}

L → Elist]	{L.place = newtemp; L.offset = newtemp; EMIT (L.place ':=' c(Elist.array)); EMIT (L.offset ':=' Elist.place '*' width(Elist.array); }
L → id	{L.place = lookup(id.name); L.offset = null;

Elist → Elist, E	{t := newtemp;
	m := Elist1.ndim + 1;
	EMIT (t ':=' Elist1.place '*' limit(Elist1.array, m));
	EMIT (t, ':=' t '+' E.place);
	Elist.array = Elist1.array;
	Elist.place := t;
	Elist.ndim := m;
	}

Elist → id[E	{Elist.array := lookup(id.name); Elist.place := E.place Elist.ndim := 1; }

Where:

ndim denotes the number of dimensions.

limit(array, i) function returns the upper limit along with the dimension of array

width(array) returns the number of byte for one element of array.

Procedures Call Translation

Procedures call

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

When a procedure call occurs then space is allocated for activation record.
Evaluate the argument of the called procedure.
Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
Save the state of the calling procedure so that it can resume execution after the call.
Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
Finally generate a jump to the beginning of the code for the called procedure.
Let us consider a grammar for a simple procedure call statement

S → **call id(**Elist)
Elist → Elist, E
Elist → **E**

A suitable transition scheme for procedure call would be:

Procedures Call Translation

Production Rule	Semantic Action
S → call id(Elist)	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)
Elist → Elist, E	append E.PLACE to the end of QUEUE
Elist → E	initialize QUEUE to contain only E.PLACE

Queue is used to store the list of parameters in the procedure call.

Declarations Translations

Production rule	Semantic action
D → integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D → real, id	ENTER (id.PLACE, real) D.ATTR = real
D → D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

ENTER is used to make the entry into symbol table and **ATTR** is used to trace the data type.

CASE Statements

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows:

```
   switch E
          begin
                  case V1: S1
                  case V2: S2
                  .
                  .
                  .
case Vn-1: Sn-1
default: Sn
          end
```

The translation scheme for this shown below:

```
Code to evaluate E into T
goto TEST
          L1:       code for S1
                    goto NEXT
          L2:       code for S2
                    goto NEXT
                    .
                    .
                    .
          Ln-1:     code for Sn-1
                    goto NEXT
          Ln:       code for Sn
goto NEXT
          TEST:      if T = V1 goto L1
                     if T = V2goto L2
                     .
                     .
                     .
                     if T = Vn-1 goto Ln-1
                     goto
NEXT:
```

When switch keyword is seen then a new temporary T and two new labels test and next are generated.
When the case keyword occurs then for each case keyword, a new label Li is created and entered into the symbol table.
The value of Vi of each case constant and a pointer to this symbol-table entry are placed on a stack.

Procedures Call Translation

Procedures call

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

When a procedure call occurs then space is allocated for activation record.
Evaluate the argument of the called procedure.
Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
Save the state of the calling procedure so that it can resume execution after the call.
Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
Finally generate a jump to the beginning of the code for the called procedure.
Let us consider a grammar for a simple procedure call statement

S → **call id(**Elist)
Elist → Elist, E
Elist → **E**

A suitable transition scheme for procedure call would be:

Storage Allocation

- The different ways to allocate memory are:

- Static storage allocation
- Stack storage allocation
- Heap storage allocation

Static Storage Allocation

- In static allocation, names are bound to storage locations.
- If memory is created at compile time then the memory will be created in static area and only once.
- Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
- Another drawback is restriction of the recursion procedure.

Stack Storage Allocation

- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.

It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

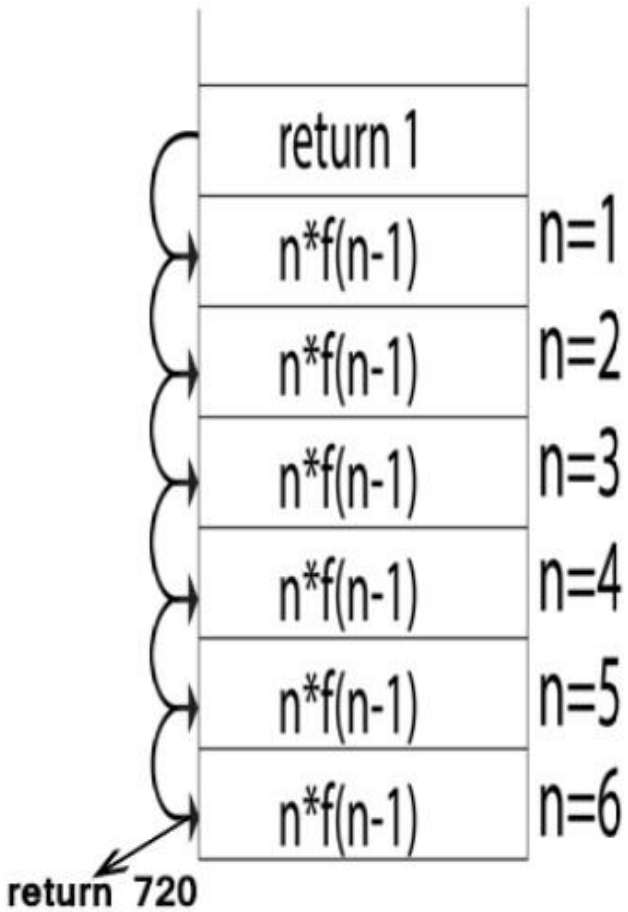
Heap Storage Allocation

Heap allocation is the most flexible allocation scheme.
Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
Heap storage allocation supports the recursion process.

Example:

```
fact (int n)
{
  if (n<=1)
     return 1;
  else
     return (n * fact(n-1));
}
fact (6)
```

The dynamic allocation is as follows:



Symbol Table Management

•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.

•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.

Symbol Table Management

•A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

•<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.

Symbol Table Management

•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:

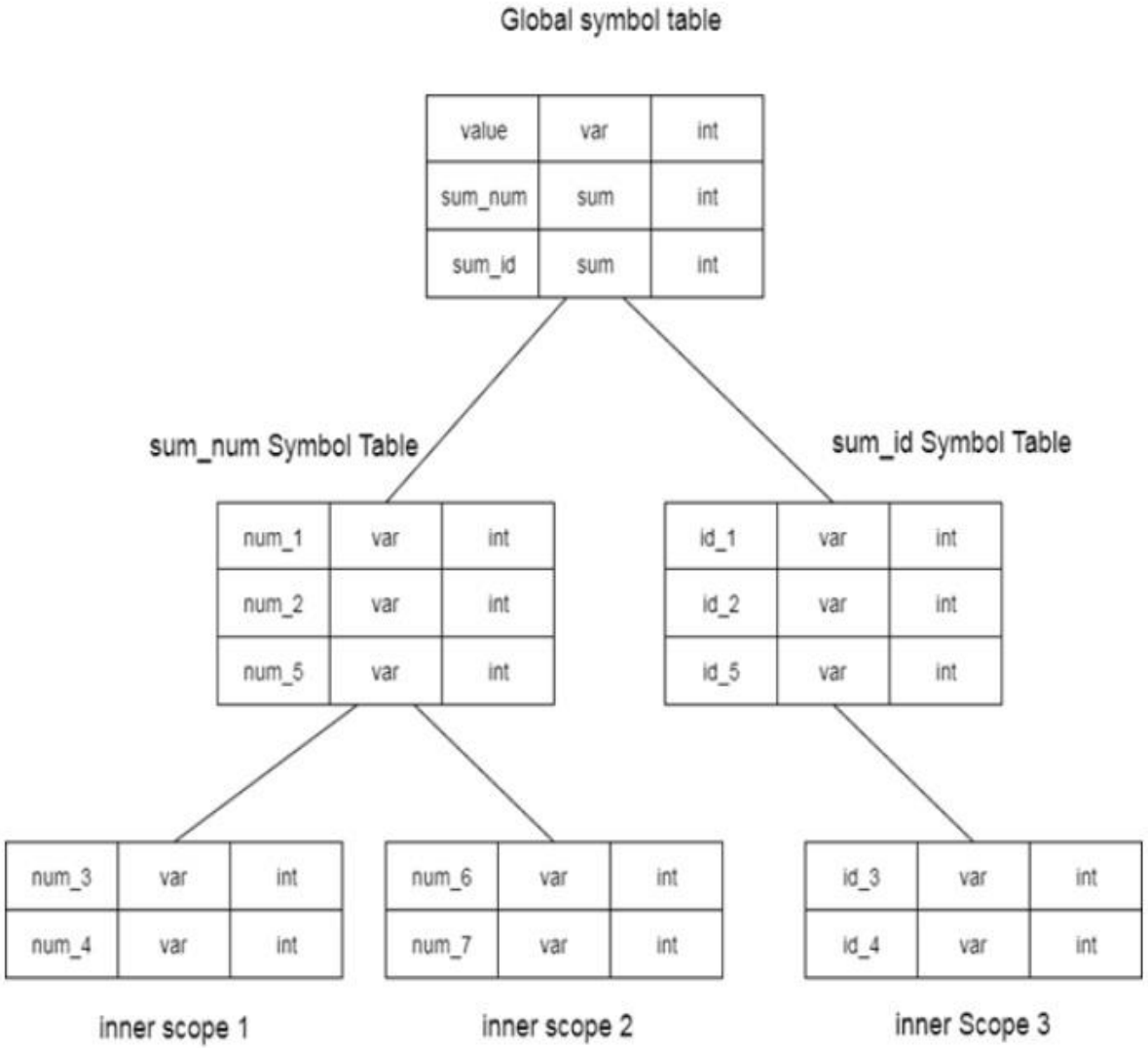
•lookup (symbol)
•This format is varies according to the programming language.

Data Structure of Symbol table

A compiler contains two type of symbol table: global symbol table and scope symbol table.Global symbol table can be accessed by all the procedures and scope symbol table. The scope of a name and symbol table is arranged in the hierarchy structure as shown below:

```
int value=10;

void sum_num()
   {
     int num_1;
     int num_2;

        {
         int num_3;
         int num_4;
        }

     int num_5;

        {
         int_num 6;
         int_num 7;
        }
   }

Void sum_id
   {
     int id_1;
     int id_2;

        {
         int id_3;
         int id_4;
        }

     int num_5;
   }
```

The above grammar can be represented in a hierarchical data structure of symbol tables:

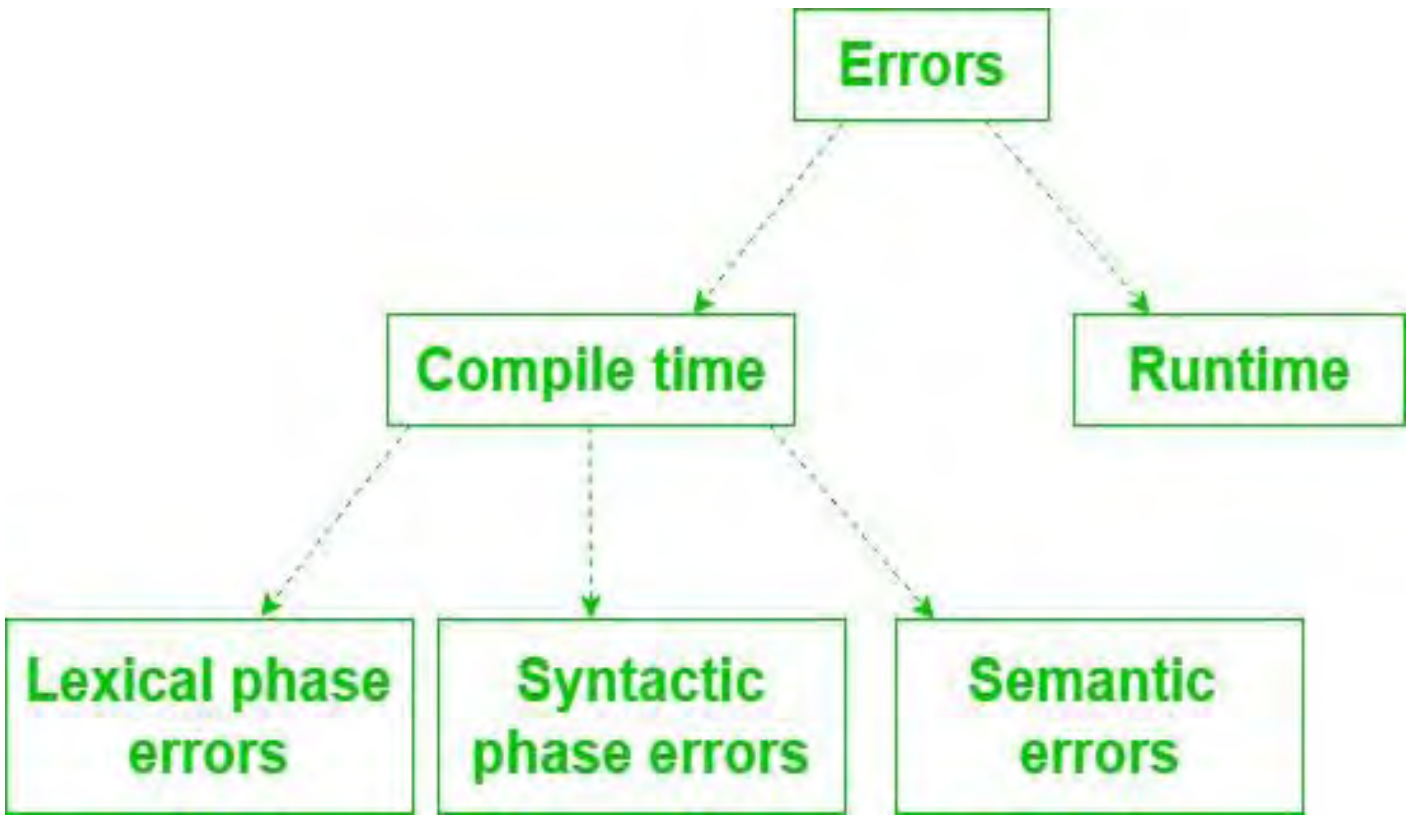


The global symbol table contains one global variable and two procedure names. The name mentioned in the sum_num table is not available for sum_id and its child tables.

Error Detection and Recovery in Compiler

- 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 PHASE ERRORS

These errors are detected during the lexical analysis phase. Typical lexical errors are

Exceeding length of identifier or numeric constants.
Appearance of illegal characters
Unmatched string

Example 1 : printf("Geeksforgeeks");$
This is a lexical error since an illegal character $ appears at the end of statement.

Example 2 : This is a comment */

This is an lexical error since end of comment is present but beginning is not present.

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 errors

Syntactic Phase Errors

These errors are detected during syntax analysis phase. Typical syntax errors are

Errors in structure
Missing operator
Misspelled keywords
Unbalanced parenthesis
Example : swicth(ch)

```
        {
          .......
          .......
        }
```

The keyword switch is incorrectly written as swicth. Hence, **“Unidentified keyword/identifier”** error occurs.

Syntactic Phase Errors

Error recovery:

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 deli-meters such as ; or }

Advantage is that its 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

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 allow the parser to parse ahead.
The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.
While performing correction, atmost care should be taken for not going in infinite loop.

Disadvantage is that it finds difficult to handle situations where actual error occured before point of detection.

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 its difficult to maintain.

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 Errors

These errors are detected during semantic analysis phase. Typical semantic errors are

Incompatible type of operands
Undeclared variables
Not matching of actual arguments with formal one
Example : int a[10], b;

```
.....
.....
a = b;
```

It generates a semantic error because of an incompatible type of a and b.

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.

Code Optimization(Introduction)

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.
- Efforts for an optimized code can be made at various levels of compiling the process.
- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.
- Optimization can be categorized broadly into two types : machine independent and machine dependent.

Code Optimization

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
  item = 10;
  value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
  value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

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 memory hierarchy.

Loop Optimization

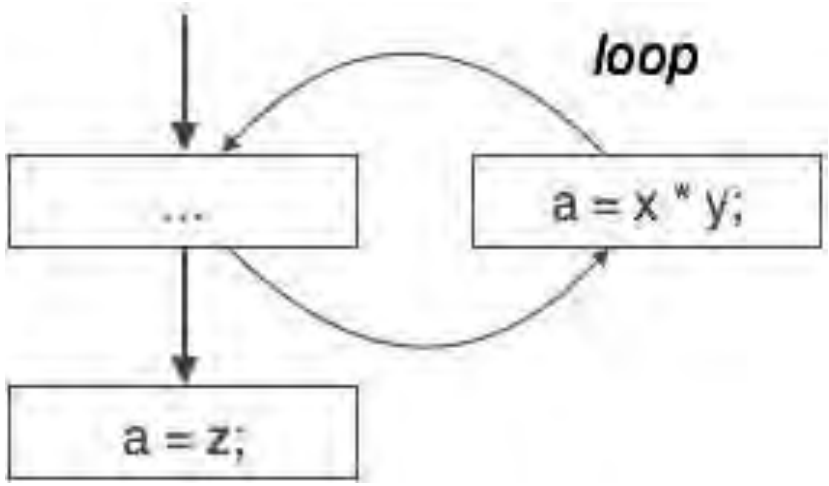
Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

Invariant code : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.

Induction analysis : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.

Strength reduction : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.



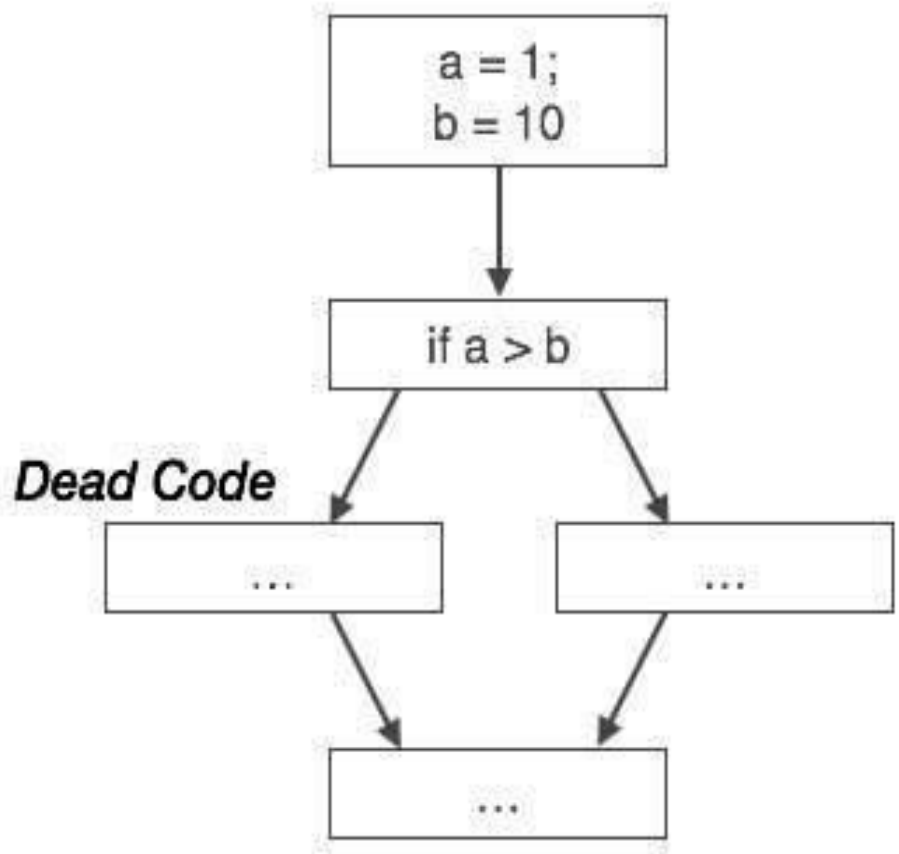
Dead Code Elimination

Dead-code Elimination

Dead code is one or more than one code statements, which are:

Either never executed or unreachable, Or if executed, their output is never used.
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code: There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



Dead Code Elimination

Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands. whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example, Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
 {
  a = y OP z;
  }
else
{
  ...
}
c = y OP z;
```

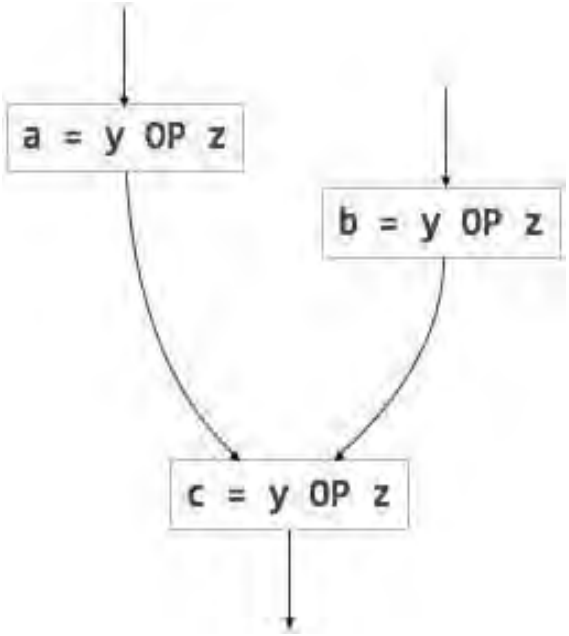
We assume that the values of operands (y and z) are not changed from assignment of variable a to variable c. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
else
{
  ...
  tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

Dead Code Elimination

Partial Redundancy



Code Generation

Code Generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.
- We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

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.

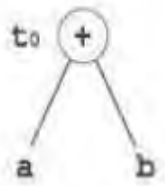
- Interior nodes represent operators.

- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

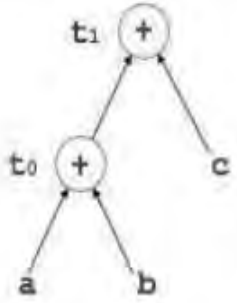
Code Generation (DAG)

Example:

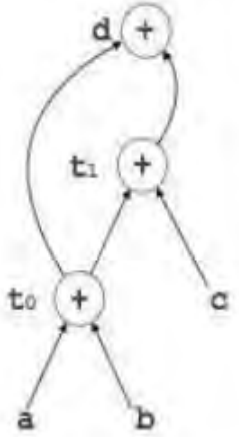
```
t0 = a + b
t1 = t0 + c
d = t0 + t1
```



[$t_0 = a + b$]



[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

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:

```
int add_ten(int x)
  {
  int y, z;
  y = 10;
  z = x + y;
  return z;
  }
int add_ten(int x)
  {
  int y;
  y = 10;
  y = x + y;
  return y;
  }
int add_ten(int x)
  {
  int y = 10;
  return x + y;
  }


int add_ten(int x)
  {
  return x + 10;
  }
```

Peephole Optimization

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 accidently written a piece of code that can never be reached.

Example:

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

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

Flow 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 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 << 1, which involves only one left shift. Though the output of a * a and a2 is same, a2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results