Compiler Design

Lecture Notes

May 6, 2025

Contents

Chapter 1

Intro

1.1 Contents

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Intermediate code
5. Syntax Directed translation
6. Code optimization
7. Gate question

1.2 Books

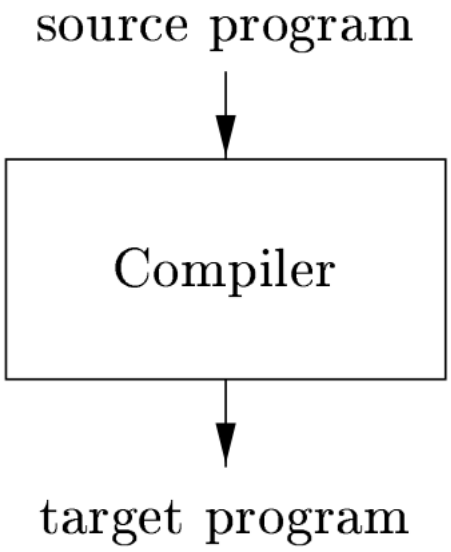
1. ullman

weightage - 4-6 marks

Chapter 2

Fundamentals

- Compiler is a translator which can translate source language into some target language.



- Program is sequence of instructions.
- Programing language is a notation by which we can specify the data operations and instructions.

- the process in which computer executes instruction is Program execution.

- Steps in program execution:

  1. preprocessing
  2. compilation
  3. assembling
  4. linking
  5. loading
  6. running

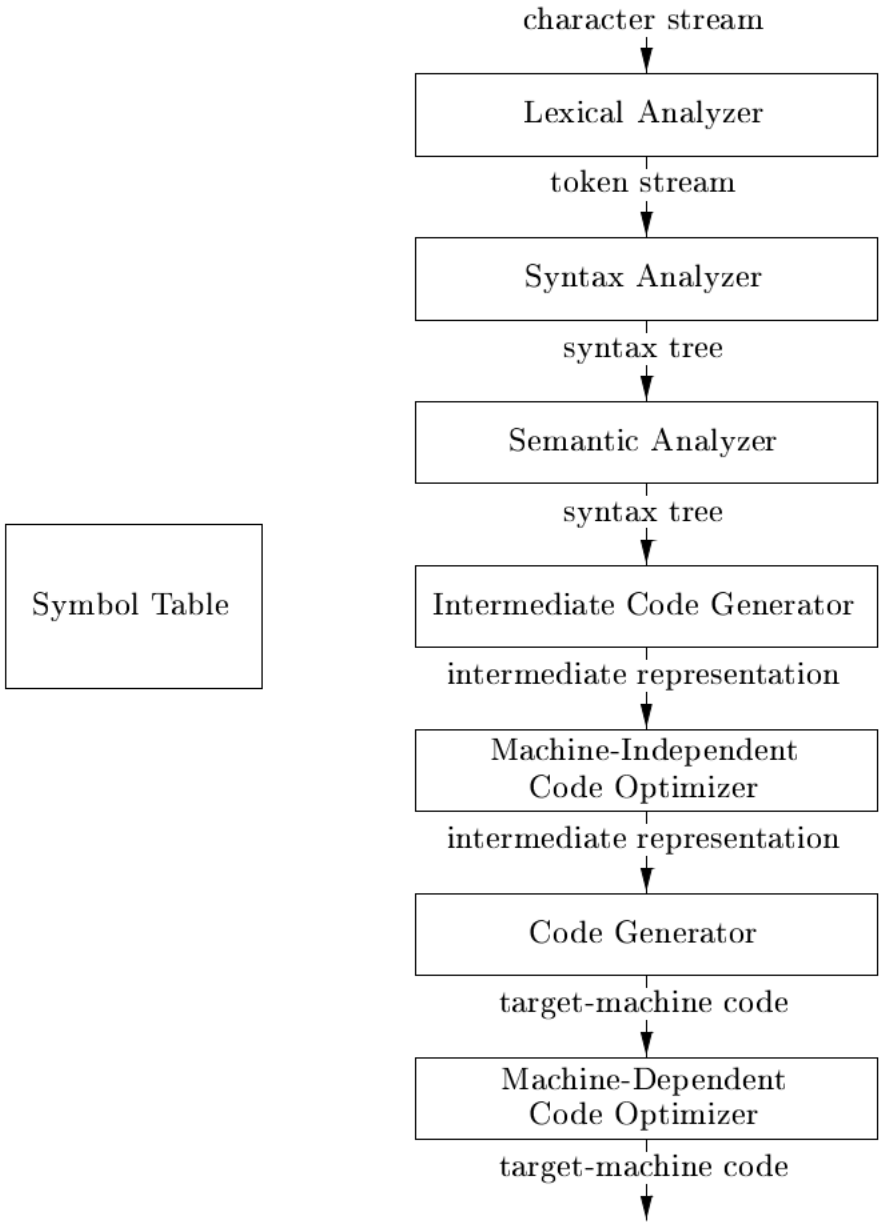
2.1 Model of compiler

compiler is composed of two phases

1. analysis : breaks the source code into constituent pieces and it creates Intermediate representation.

2. synthesis : combine constituent pieces and constructs desired target program.

2.2 Phases Of Compiler



Analysis

Lexical Analysis : it is basically a DFA which checks the spelling or keyword error, to check spelling mistakes the lexical analyzer breaks the whole programme into tokens.
Syntax Analysis : it checks the syntax error or grammatical error, by using a CFG, it parses the given programme according to CFG by either LMD or RMD. The machine which does this task is called parser.
Semantic Analysis : it checks for errors in the logic or meaning of a programme. Parser is also used for semantic analysis.
High-level language : a language in which one statement has multiple operations.
Low-level language : a language in which one statement has one operations.
Intermediate Code : compiler converts our program into some low level language which is it's intermediate code, Intermediate Code is generally a low level language. Intermediate code is easy for compiler to optimize and code generation. there are three types of intermediate code

1. Syntax tree
2. postfix notation
3. three address code further divided into three types
   (a) Quadruple
   (b) Triple
   (c) Indirect

Synthesis
Code Optimization : techniques of code optimization

1. Code Motion
   (a) Loop unroll
   (b) Loop invariant construct
2. Strength reduction
3. Common subexpression elimination
4. Dead code & unreachable code

Code generation : not in syllabus.

2.2.1 Symbol table

It is not a phase of compiler but it is it,s significant part without which it will not function, it is shared between all the phases of compiler, it is metadata for the program.

The symbol table stores important data like variable type class etc. which is further used by other phases.

2.2.2 Error handler

neither a phase nor a part of compiler but a third party software which is used for user convenience for reporting errors and shared by all phases.

there are no chances of error in synthesis phase it is used for reporting OS errors.

Chapter 3

Syntax Analysis

The study of syntax analysis is devoted to the construction of parsers.

parser is a machine that checks for syntax errors in a program according to the grammar.

There are two types of parsers

1. Top-down parser LL(1) or non recursive descent parser or pridictive parser
2. bottom-up parser or shift-reduce parser or LR parsers,(operator precedence parsers are bottom-up) further divided into
   (a) LR(0)
   (b) SLR(1)
   (c) CLR(1) - most powerful
   (d) LALR(1)

   all these parser are table driven parser.

3.1 Top-down parsers

Top down parsers uses left most derivation.

It follows pre-order and DFS technique.

specifically LL(1) parser have no backtracking and is non recursive.

uses stack explicitly

3.1.1 Algo to construct LL(1) parsers

1. Remove left recursion if required.
2. Left factor the grammar if required.
3. find first & follow sets.
4. construct the parsing table.

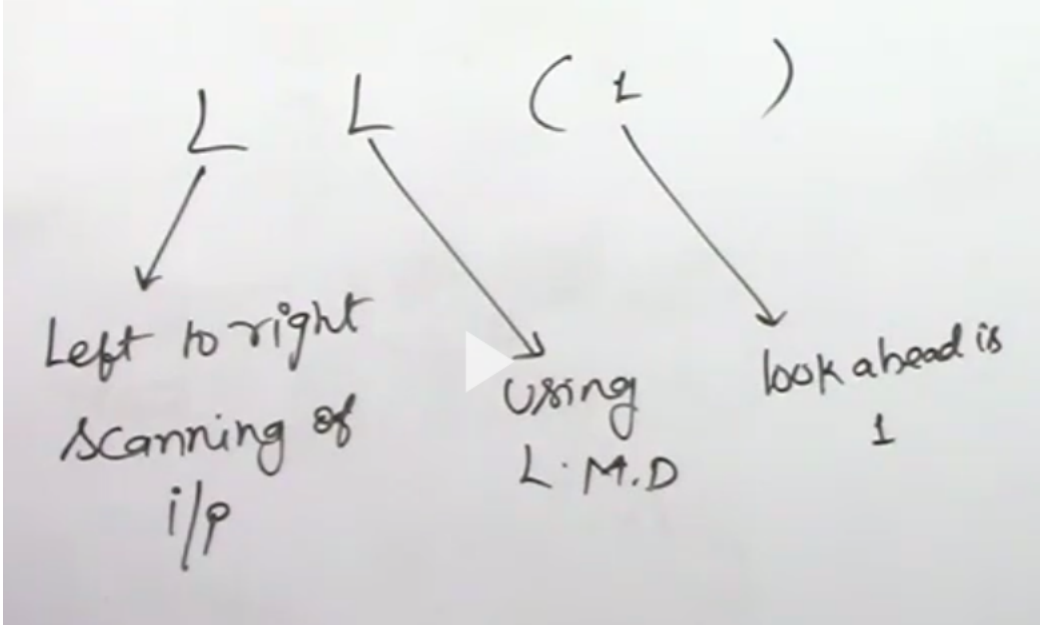
Given a grammar if it is possible to construct a LL(1) parser the grammar is LL(1).

the LL(1) parser sees one character of a string at a time this character is called look ahead symbol the 1 in LL(1) signifies this fact.

when a production in a grammar has same prefix on two or more of its derivation removing those is called left factoring.

to avoid backtracking we have to left factor the grammar.

it is called prefictive because we don,t have to backtracking.



3.2 Removal of left recursion

consider a grammar $S \rightarrow Sa|SSA|ab|ABC|ABS|ASA|BSS|aS|Sab|aSb$

divide the grammar into two parts one that does not contalin left recursion one that contains.

$S \rightarrow abX|ABCX|ABSX|ASAX|BSSX|aSX|aSbX$

$X \rightarrow aX|SAX|abX|\epsilon$

3.2.1 Indirect left recursion

consider a grammar

$A \rightarrow Ba|ab$

$B \rightarrow Cb|bc$

$C \rightarrow Ac|ca$

this grammar has an indirect left recursion.

the grammar is equivalent to

$A \rightarrow Cba|bca|ab$

$C \rightarrow Ac|ca$

which is equivalent to

$A \rightarrow Acba|caba|bca|cb$

this ha left recursion to be resolved

3.3 Removal of left factor

consider

$A \rightarrow aAb|aA|b$

two derivation cannot have same prefix here aAb and aA has a as common prefix.resolve this as follows

$A \rightarrow aX|b$

$X \rightarrow Ab|A$

continue like this and ultimate all factors will be removed

or directly

considering aA as prefix

$A \rightarrow aAX|b$

$X \rightarrow b|\epsilon$

3.4 First and Follow sets

3.4.1 First set

first of a variable is the set of leftmost terminals of all generated string of that grammar.

consider the following grammar

$S \rightarrow AB|CD$

$A \rightarrow aA|a$

$B \rightarrow bB|b$

$C \rightarrow cC|c$

$D \rightarrow dD|d$

the first set is as follows

variable	First
S	a,c
A	a
B	b
C	c
D	d

consider another grammar

$S \rightarrow AB|CD$

$A \rightarrow aA|a|\epsilon$

$B \rightarrow bB|b$

$C \rightarrow cC|c$

$D \rightarrow dD|d$

the first set is as follows

variable	First
S	a,b,c
A	a ,ϵ
B	b
C	c
D	d

existence of epsilon matters only if it comes alone

3.4.2 Follow set

Rules

1. Include $ in follow of start variable

2. if production is of type $A \to \alpha B\beta$ where $\alpha and \beta$ are strings of grammar symbols Follow(B)=First(β) (copy first of β in follow of B)
3. if production is of type $A \to \alpha B$ or $A \to \alpha B\beta$ where $\alpha and \beta$ are strings of grammar symbols and $\beta \overset{*}{\to} \epsilon$ then Follow(B)=Follow(A) (copy follow of A in follow of B)

imp. points

- ϵ is not used in follow of any variable
- ϵ is used oly in first set if required
- first and follow sets will never contain variables it only contains terminals
- $ is used only in follow sets of variables.
- ϵ is not included in follow set.

3.5 Construction of Parsing table

once left recursion is removed and we have constructed first and follow sets we are ready to construct parsing table.

consider the following grammar
$E \to E + T|T$
$T \to T * F|F$
$F \to (E)|id$
removing the left recursion and left factor this becomes
$E \to TE'$
$E' \to +TE'|\epsilon$
$T \to FT'$
$T' \to *FT'|\epsilon$
$F \to (E)|id$
The first and follow sets are as follows

	first	follow
E	(,id	$,)
E'	+,ϵ	$,)
T	(,id	$,),+
T'	*,ϵ	$,),+
F	(,id	*,$,),+

Rules for construction of parsing table.

- The column of the table contains all terminals including $ and excluding ϵ.
- The row of the table contains all non-terminals.
- A cell of the table will be filled if the non-terminal corresponding to the cell has the terminal corresponding to that cell. And that cell will be filled with the production which led that terminal to be included in first set.
- If the first set of terminal contains ϵ in first se, then the cell corresponding to the terminal in the follow of that non-terminal will also be filled but with production which led to ϵ being included in it's first cell.
- All the cell left empty will be filled with 'Error'.

The parsing table of above grammar is as shown

	+	*	(	)	id	$
E	Error	Error	$E \rightarrow TE'$	Error	$E \rightarrow TE'$	Error
E'	$E' \rightarrow TE'$	Error	Error	$E' \rightarrow \epsilon$	Error	$E' \rightarrow \epsilon$
T	Error	Error	$T \rightarrow FT'$	Error	$T \rightarrow FT'$	Error
T'	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	Error	$T' \rightarrow \epsilon$	Error	$T' \rightarrow \epsilon$
F	Error	Error	$F \rightarrow (E)$	Error	$F \rightarrow id$	Error

Imp. points.

- if a cell potentially has two or more entries LL(1) parsing table is not possible.
- The parser uses the table and a stack to parse a string.

3.6 How to check whether the grammar is LL(1)

If the productions is of type $A \rightarrow \alpha_1|\alpha_2|\alpha_3$
$First(\alpha_1) \cap First(\alpha_2) = \phi$
$First(\alpha_1) \cap First(\alpha_3) = \phi$
$First(\alpha_2) \cap First(\alpha_3) = \phi$
If the productions is of type $A \rightarrow \alpha_1|\alpha_2|\alpha_3|\epsilon$
$First(\alpha_1) \cap First(\alpha_2) = \phi$
$First(\alpha_1) \cap First(\alpha_3) = \phi$
$First(\alpha_2) \cap First(\alpha_3) = \phi$
$Follow(A) \cap First(\alpha_1) = \phi$
$Follow(A) \cap First(\alpha_2) = \phi$
$Follow(A) \cap First(\alpha_3) = \phi$
if for all productions of grammar this rules hold then the LL(1) parser is possible.

3.7 Bottom-Up Parsers

- Bottom-up parsers uses LMD in reverse
- They are also known LR parsers or SR(shift reduce) parsers.
- these parses can work with left recursion as well as left factor.

3.7.1 Algorithm to construct Bottom-up parsers

1. Expand the grammar (write the grammar without the use of '|')
2. Number all the productions (write serial no. for expanded grammar)
3. Augment the grammar (define a new start variable producing the original start variable)
4. Find first and follow sets except the augmenting variable.
5. Find LR(0) canonical item set
6. construct the table.

Consider The following grammar

$E \to E + T|T$
$T \to T * F|F$
$F \to (E)|id$
Expanding we get
$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to id$
Numbering this we get

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

Augmenting this we get
$E^{`} \to E$

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

3.7.2 Finding LR(0) item set and Sets constructionb

it is nothing but a technique used by bottom-up parser LR(0) canonical item set is also known as LR(0) DFA.

See lectures section 2 13-15/21

Imp. Points

- If item set contains at least one shift corresponding to terminals and at least one reduction then the state is inadequate state.
- If a cannonical item set has two or more reduction then the state. is inadequate state
- If a state only shift operations then it is not inadequate state.
- If we get at least one inadequate state for a grammar then the grammar can never be LR(0) but can be SLR(1) [why??].
- Two shift entries in one cel is not possible.
- we may have more than one reduction entries in a cell (reduce reduce conflict).
- we may have a single shift and more than one reduction entries in a cell (Shift reduce conflict).
- shift shift conflict can never arise.

3.8 SLR

- SLR(1) $\rightarrow$ Simple Left to right scanning of i/p using Reverse RMD Lookahead 1
- LALR(1) Look Ahead Left to right i/p using Reverse RMD Look ahead 1
- LR(0) $\rightarrow$ Left to right scanning of i/p Reverse RMD Lookahead 0
- CLR(1) Cannonical Left to right scaning of i/p reverse RMD Lookahead is 1

- CLR(1) Cannonical Left to right scaning of i/p reverse RMD Lookahead is 1

By using LR(0) cnnonical item set we can construct two parsers LR(0) and SLR(1)

By using LR(1) cnnonical item set we can construct two parsers CLR(1) and LALR(1)

3.9 CLR(1) aka LR(1) parsers

3.9.1 LR(1) canonical item set

3.9.2 LR(n) item set

consider

$S \to A.ab, \#|abc|12$

the above is combination of three item sets

$S \to A.ab, \#$

$S \to A.ab, abc$

$S \to A.ab, 12$

here the compiler will have to look for atmost 3(for abc) char so it is LR(3) item set.

3.9.3 Detail calculation of LR(1) item set

see Lecture 17

Imp.Points

- Lookahead is calculated this way
  - we start with \$ as look ahead
    $S` \to .SAB, \$$
    $S \to \alpha_1, FIRST(AB\$)$
    $S \to \alpha_2, FIRST(AB\$)$
  - if
    $S` \to .SAB, \$|\#$
    $S \to \alpha_1, FIRST(AB\$) \bigcup FIRST(AB\#)$
    $S \to \alpha_2, FIRST(AB\$) \bigcup FIRST(AB\#)$

- the difference between entries in LR(0) and LR(1) is that here we do reduction entries in Lookahead symbols (bookish method is to make reduction entries in the intersection of follow(of the variable which produced the reduction)and lookahead).

- it is most powerful parser

- Ignoring lookahead then some itemset will be equal here we can merge them by taking union of lookaheads that will make it a procedure for LALR(1) parser only if there are no conflicts.

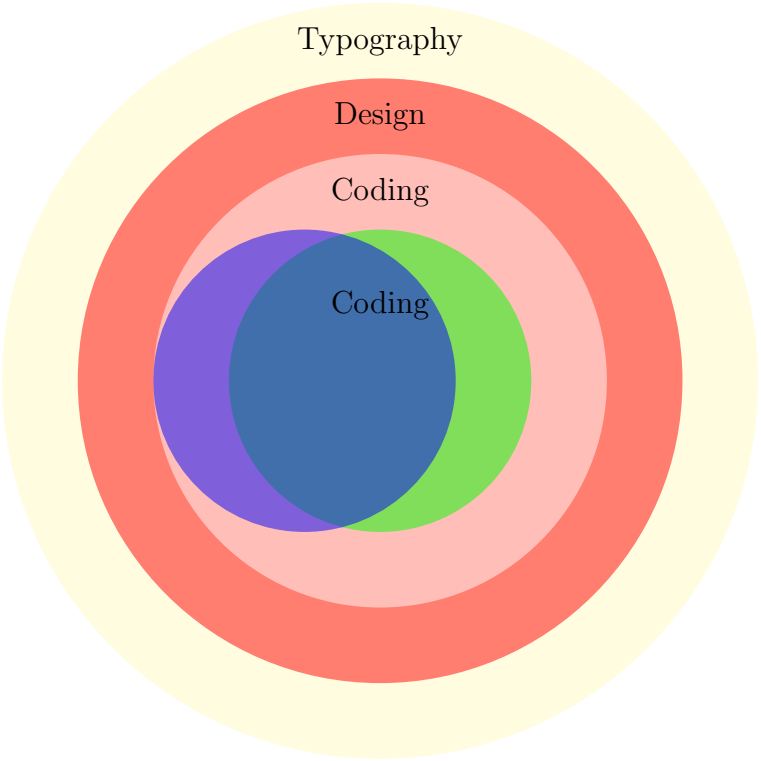
- If there is no conflict CLR(1) table there is a possibility that conflict may arise after merging.

- no. of states of itemset in SLR(1)=LALR(1)<=CLR(1)

- if there is no SR conflict in CLR(1) then SR conflict will never occur in LALR(1) after merging

- if there is no R conflivt in CLr(1) RR conflict may occur in LLR(1) after merging

-



3.10 Algorithms

todo lecture clr and lalr
imp.point

- A handle is a substring in right senetentialform that matches the body of production in grammar and whose reeduction represents one stem in reverse RMD.
- YACC resolves SR conflict in favour of Shift.
- all parser are O(polynimial).

3.11 Operator grammar

this grammar follows the following rules

- the grammar should not have two variable side by side in its productions $s \rightarrow vv$ not allowed.
- no ϵ productions.
- operator grammar is possible without operators.

Chapter 4

Syntax Directed Translation

There are two types of Syntax Directed Translation

- Syntax directed Definition : when we write the simantic rules at the end of productions.
- Translation schemes : when we dont care where we write rules (every SDD is translation scheme but not vice versa)

There are two types of attribute

- Synthesised : an attribute whose value depends on any of its children
- Inherited : an attribute whose value depends on its sibling or its parent or both.(in SDT we use restricted inheritance that is a node can inherit only from its left sibling this is called L-attributed).

SDT based on attributes is of two types

- S-attribute : SDT contains all Synthesised attribute
- L-attribute : SDT we use Inherited attribute but restricted inheritance that is a node can inherit only from its left sibling, top down left to right approach will be easy to evaluete.

Annonated Grammar

when we use some simantic rules along with the Grammar it is called annotated grammar

eg:E $\rightarrow$ E#T E.value=E.value*T.value

while parsing we perform more yhan one task at a time so this annotated Grammar enables us to do some calculation while parsing itself.

Chapter 5

Intermediate Code generation

Intermediate code

1. Linear form

   (a) post-fix notation

   (b) 3-address codes : a low level statement where there are atmost 3 memory references. three way to implement this in a program

   - quadruple

operator	operand 1	operand 2	result

     – order of statement can change

     – space wasted

   - triple

operator	operand 1	operand 2

     – order of statement can not change

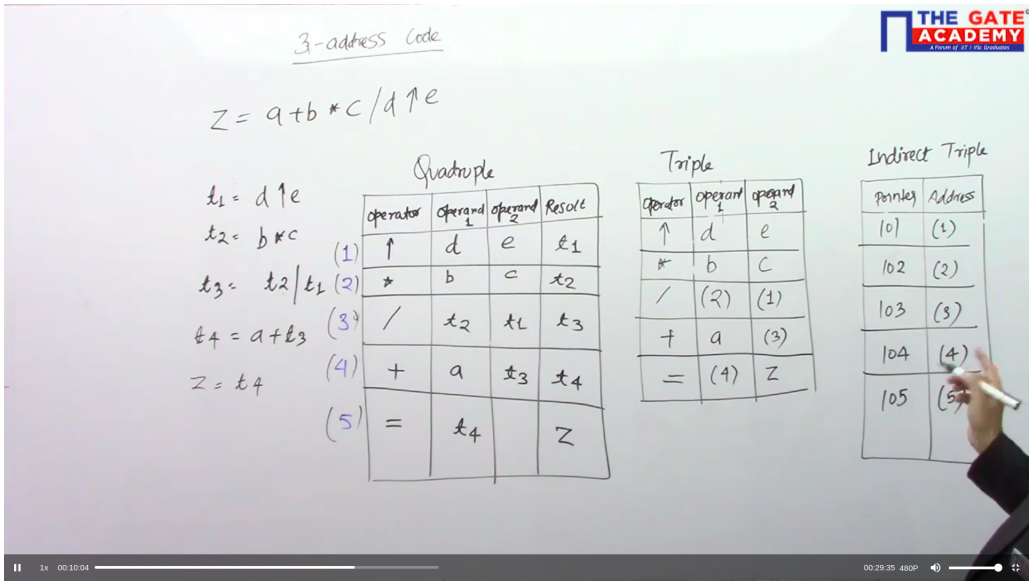
     – space not wasted

   - indirect triple

pointer to address	address

     – order of statement can change

     – space not wasted bit complex



2. tree from

   (a) Syntax tree : abstract syntax tree is also known as parse tree syntax tree and parse tree are different.

   (b) Directed acyclic graph : it is used in common sub expression elimination code optimization as well as intermediate code.

5.1 Converting high level code to intermediate code

1. if the high level code uses three address use

2. if not then break it down

consider the following

```
if(a<b){
     t=1;
}else{
     t=0
}
rest of the program
```

intermediate 3 address code

1. if(a¡b) goto 4
2. t=0
3. goto 5
4. t=1
5. rest of the programe

leaving goto lables as empty and filling them later is called back patching.

by looking at three address code we cannot predict high level program to detect the loop in three address code we have to make control flow graph of the three address code.

5.2 Control flow graph for detecting a loop

To detect loops we divide three address codes into some basic blocks.

Basic blocks : it is a sequence of three address code where controls enters from first statement and exits from last statement. there should be no conditional [if(a>b)goto 7] or unconditional [goto 6] jumps within basic block.

to find basic blocks we find leaders.

Finding leaders

- first three address code statement is a leader
- the target of a jump is a leader.
- three address code statement just below the jump is a leader.
- the jumps are itself a leader.

Finding Basic block start from a leader and stop at the statement just above the next leader, theses statement forms a basic block.

make a control flow graph (lecture)

if control flow graph contains cycle there is a loop.

Chapter 6

Code optimization

Compiler uses code optimization tecqniques only to reduce no. of line in code to save cpu cycles.

there are many optimization tecqniques but we discuss a hardware independent tecqniques called common subexpression elimination.

to elimante common subexpression compiler uses DAG.

6.1 common subexpression elimination

same subexpression being compute d multiple times.

6.1.1 common subexpression elimination using DAG

see sec 5 lecture 1/1

6.2 loop invarient computation

some same computation taking place in a loop .

6.3 strength reduction

* and / are replaced wit + and - respectively because they are more strong

6.4 dead code elimination

some code which is not required like a variable never used statements after return

Chapter 7

Lexical Analyzer

It is also known as scanner.

Typical tas of scanner

- counts the number of line in program.
- it ignores comment lines.
- it treats tabs and spaces as blanks.
- scanner converts the program into stream of tokens.
- it uses DFA to check the valid tokens.
- it reports lexical or spelling errors in program.
- it recognizes the reserved key words.
- it finds the variables in the program.
- it is the first phase to identify variables and write it into symbol table.
- LEX is a lexical analyzer generator it takes regular expression as input.

Token : it is sequence of meaningful characters eg: keywords, operator, variables/identifiers, constants etc.

Lexeme : sequence of characters matched by some pattern or some token.

Pattern :a rule describe Lexeme.

lexical analyzer chooses a substring using ***pattern*** *and call it* ***Lexeme*** *then passes it into dfa if it passes it becomes a* ***token****.*

it considers a string as one token eg: "i=%d,&i,%x"

Inline expansion : it is code optimization tecqnoque used by compiler, the compiler replaces the function call with the body of the called function it is same as macro expansion but inline expansion is done during compilation and macro expansion is done before compilation during preprocessing. also known as function inline

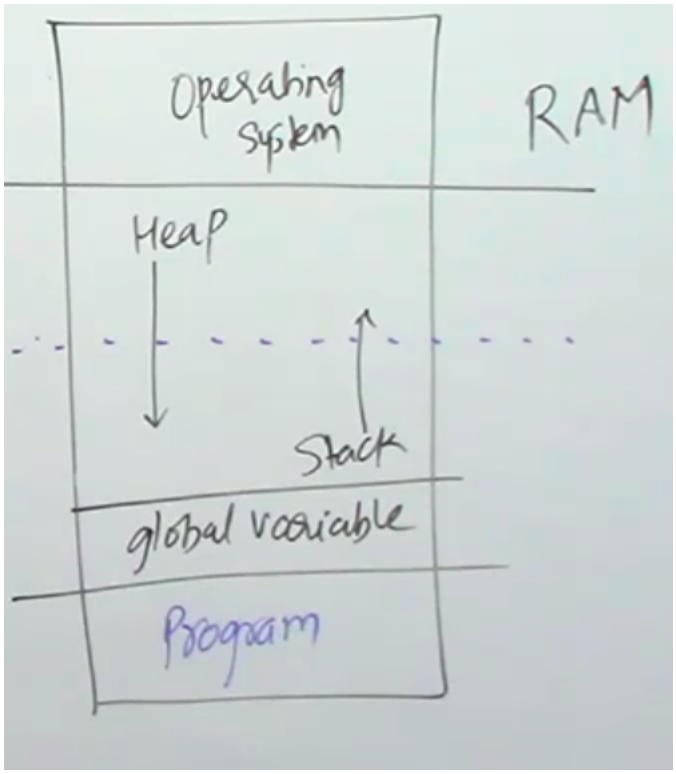
every programing language is written by CSG

we need asembler to convert the obj code in assembly language to machine code. if compiler does not gives machine code directly.

compiler	interpreter
it uses whole program as input	it executes line by line

7.1 Runtime environments

the services requested by os is the run time environment for a program.



7.1.1 Static and Dynamic Scoping

TODO

7.1.2 Static and Dynamic memory allocation

Static memory allocation : variables get allocated permanently and allocation is done before program execution but during compilation,it uses the data structure for implementing static allocation and it is less efficient than dynamic memory allocation there is memory reusability in static allocation.

Dynamic memory allocation : in this case variables get allocated only after compilation during execution. it use the data structure heap for implementing dynamic memory allocation, more efficient then static allocation. In dynamic memory allocation we can reuse the memory and we can free the memory when not required.

7.1.3 Activation record

a data structure containing the important state information for a particular instance of a function call, it may exist entirely in the memory or or partially be in the registers.