CS 307

B.Tech 6th Sem

CSE

COMPILER DESIGN

Subhojit Ghimire
1912160

SYLLABUS:

1. Lexical Analysis : Tokenisation
2. Parser (Syntax Analyser) (V. Imp).
3. Semantic Analysis
4. Intermediate Code Generator (Imp):
5. Code Optimisation.

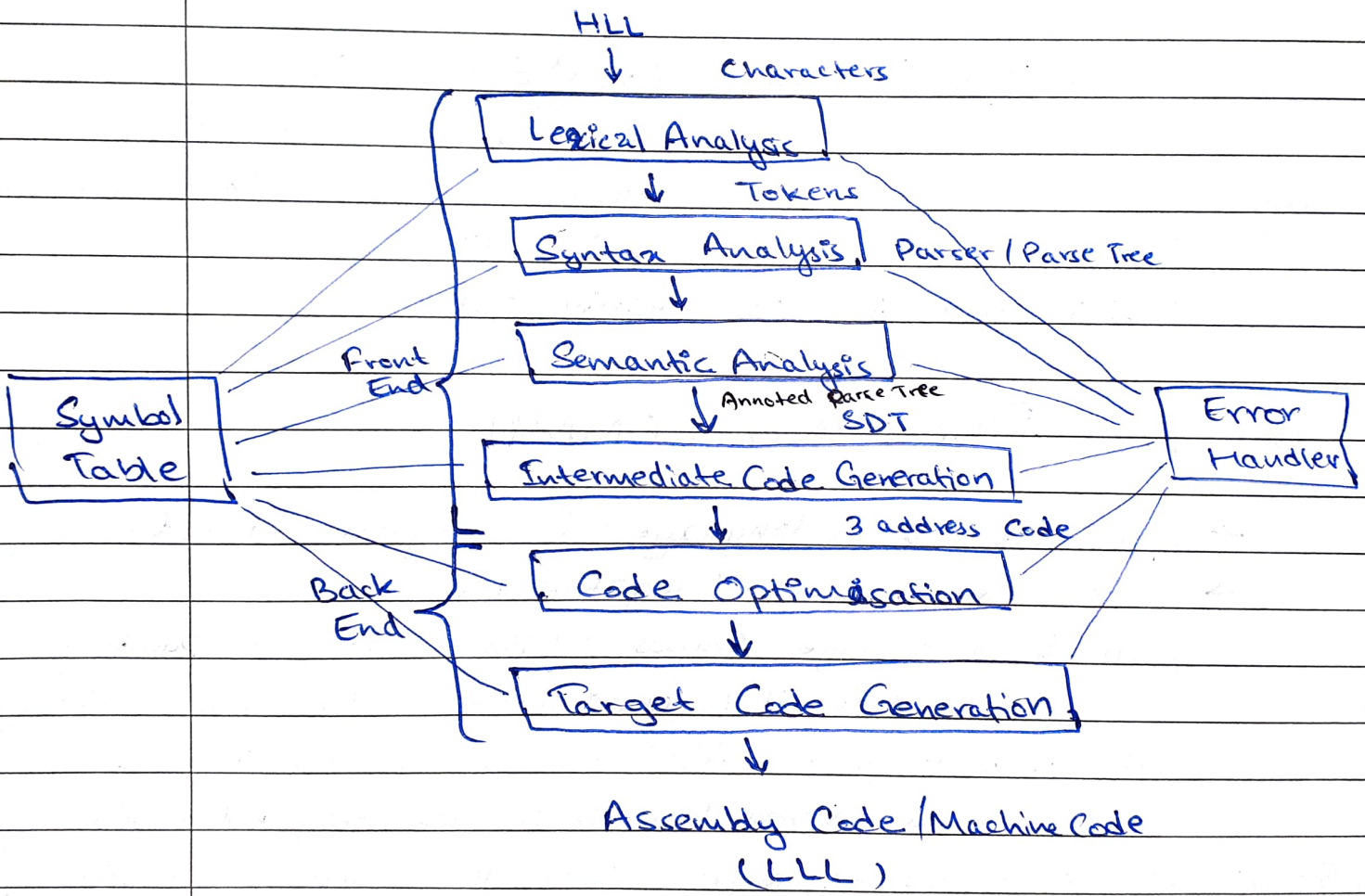


fig.: Stages in Compiler

__/__/____

* Lexical Analysis [Lexer, Tokeniser, Scanner]
  - Tokenisation
  - Give Error Message
    - Exceeding length
    - Unmatched String
    - Illegal Character
  - Eliminate Comments, White Spaces

Tokens: Identifiers, Separators, Keywords, Operators, Constants (Literals), Special Characters.

* Finding first() and follow() in any grammar:
  - first(A) contains all terminals present in first place of every string.

first (terminal) = terminal

first (ε) = ε

E.g.: first (abc) = a

first (abc|def|ε) = a, d, ε

first (ABCD) = first (A)

↳ if A = a|b|c

first (A) = a, b, c

Then; first (ABCD) = a, b, c

E.g.:
E → TE' ; first(E) = first(T) = id, (
E' → *TE' | ε ; first(E') = *, ε
T → FT' ; first (T) = first(F) = id, (
T' → ε | +FT' ; first(T') = ε, +
F → id | (E ; first (F) = id, (

(Bottom to top find value for first)

if, T → FT' | ε ; first(T) = id, (, ε

then, first(E) → first(T) = id, (, ε

As, ε (epsilon) is present, we make ~~T~~ ε and,

first (E') = *, ε

So, first (E) → id, (, *, ε

↳ follow (A) contains set of all terminals present immediately in right of 'A'

Rules: 1. follow of start symbol is $, i.e,

Follow (A) = {$}

2. S → ACD

C → a | b

Follow (A) = first (C) = {a, b}

Fo (D) = follow (S) = {$}

3. S → aSbS | bSaS | ε

* Follow never contains ε

fo (S) = $, b, a.

Example: S → ABC

B → ε

C → ε

A → DEF

follow (A) = first (B) = ε * follow cannot contain ε.

So, follow (A) = first (C) = ε * again ε, So,

follow (A) = S → A B C (ε ε) follow (S) = $

__/__/____

* Parsers:

↳ Parsing is a process of deriving string from a given grammar.

- Parsers
  - Top Down Parser
    - Recursive Decent
    - LL(1) (predictive)
  - Bottom Up Parser
    - LR(K)
      - LR(0) canonical items
        - LR(0) parsing table
        - SLR(1) parsing table
      - LR(1) canonical items.
        - LALR parsing table
        - CLR parsing table.
    - Operator Precedence

LL(1) → RHS First. (if first (RHS) = {∈}, then follow(LHS)

Self Note:

LR(0) → reducing → entire row.

SLR(1) → reducing → Follow of LHS.

CLR, LALR → reducing → look ahead terminals.

* SDT: → Top-Down → {write terminal, non-terminal, action

Bottom-Up → {write only terminal and non-terminal.

↳ during parse tree.

* Basic Block: Sequence of Intermediate codes with single entry and single exit.

__/__/____

* Types of SDT (Syntax Directed Translation)

- S-Attributed
  - Based on Synthesised Attribute
  - Use Bottom Up Parsing
  - Semantic Rules always written at rightmost position in RHS
- L-Attributed
  - Based on both Synthesised & inherited attributes
  - Top Down Parsing
  - Semantic Rules anywhere in RHS.
  - (Synthesised on Parent and "Left" Sibling only).

* Intermediate Code Generation

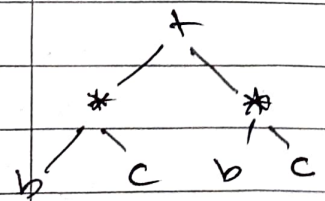
↳ Machine Independent

- Abstract Syntax Tree
- Direct Acyclic Graph.
- Postfix, Prefix.
- 3-Address Code* (v. imp).

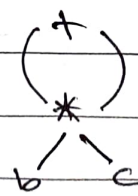
E.g.:

(b * c) + (b * c)

Syntax Tree



Acyclic Graph



Postfix

b c * b c * +

Prefix

+ * b c * b c

→ 3-Address Code:

$t_1 = b * c$

$t_2 = b * c$

$t_3 = t_1 + t_2$.

__/__/____

* Three-Address Code Representations:

1. Assignment : $x = y \text{ op } z$

$x = \text{op } y$

$x = y$

2. Jump : Conditional : if x relop y goto L

Unconditional : goto L

3. Array Assignment : $x = y[i]$

$x[i] = y$

4. Pointer, Addr. Assign : $x = \&y$

$x = *y$

* Peephole Optimisation:

1) Redundant Load and Store.
2) Strength Reduction.
3) Simplify Algebric Expression.
4) Replace Slower Instructions with faster.
5) Deadlock Elimination.

* Code Optimisation

- Platform Dependent Techniques (Platform/Machine)
  - Peephole optimisation *
  - Instruction level Parallelism
  - Data level Parallelism
  - Cache Optimisation
  - Redundant Resources.
- Platform/Machine Independent Techniques
  - Loop Optimisation *
  - Constant folding
  - Constant Propagation
  - Common Subexpression Elimination.

* Three Address Code:

$t_1 = y * z$

$t_2 = x + t_1$

Eg:. Converting $a = b + c * d$ (four addresses) to 3-Address Code.

So, $t_1 = c * d$

$t_2 = b + t_1$

$a = t_2$.

} Also called intermediate code

What is the requirement of intermediate code?

Three Address Code

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

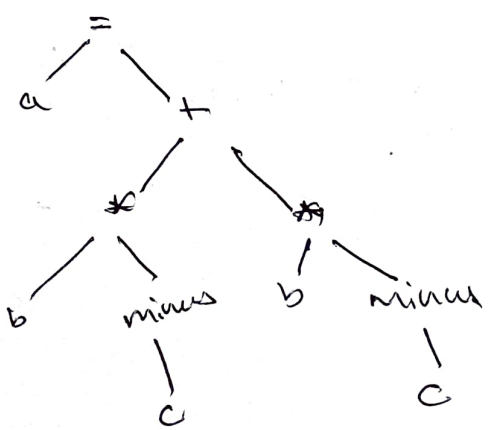
(a)

Quadruples (4 columns)

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

(b)

Syntax Tree



(c)

Triples (3 columns)

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

(d)

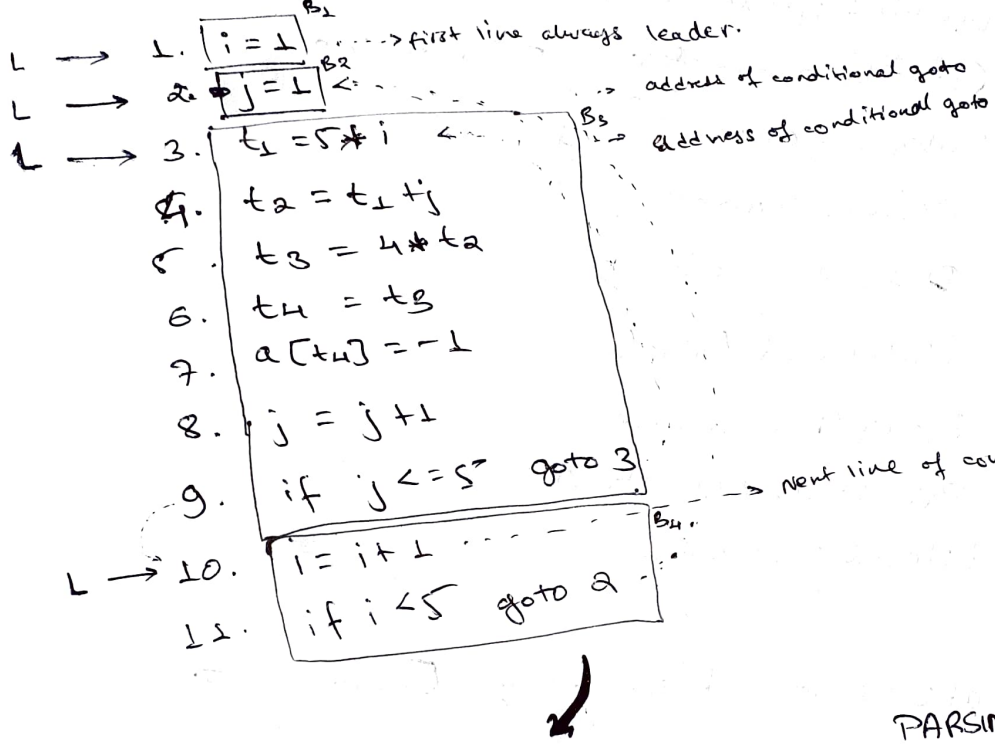
fig: Representation of :
(a)(b)(c)(d)
$a = b * -c + b * -c$

Chapter 8: Code Generation:

CS307 CD 02/05/202

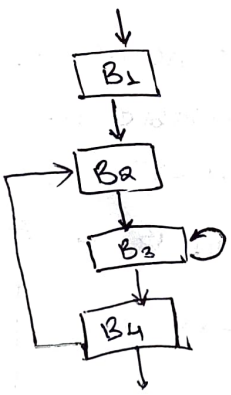
* State the steps in code Generation:
* for given line of code:
* Basic block can be of one or more structures:
* Basic properties of the basic block: Control Enter → [Basic Block] → Control Exit.
* **Leader** :(i) first line of the block. (i=1) (ii) Address of conditional/unconditional goto. (iii) Next line of conditional/unconditional goto.

- How to find leader in basic block?



4-blocks.

* Control Flow Graph:



PARSING SDT

CODE OPTIMISATION
- Platform independent code optimisation
- platform dependent code optimisation
- Peephole Optimisation
- Loop Optimisation

CONTROL FLOW GRAPH AND BASIC BLOCKS.