KULVIR SINGH
19BCE2074

(19)

1) Machine independent optimisation attempts to improve the intermediate code to get a better target code. The part of the code which is transformed here does not involve any absolute memory location or any CPU registers.
Code optimisation can be achieved using Compile Time Evaluation, Variable Propagation, Dead code compilation, code motion, induction variable and Strength Reduction.

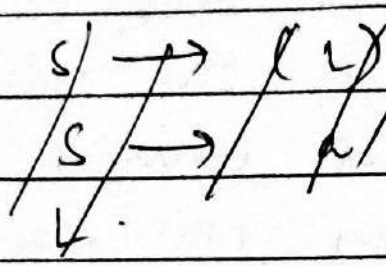
2) Global Data flow Analysis:
To efficiently optimise the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis. Certain optimisation can only be achieved by examining the entire program. It can't be achieved by examining just a portion of the program.

3) Predictive Parser —

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Removing Left Recursion :



$S \longrightarrow (L) \mid a$

$L \longrightarrow (L)L' \mid aL'$

$L' \longrightarrow , SL' \mid \epsilon$

First(S) = { (, a }

First(L) = { (, a }

First(L') = { , , ∈ }

Follow(S) = { $, , , ) }

Follow(L) = { ) }

Follow(L') = { ) }

Predictive Parser Table

| Non-Terminal | ( | ) | a | , | $ |
|---|---|---|---|---|---|
| S | $S \longrightarrow (L)$ | | $S \longrightarrow a$ | | |
| L | $L \longrightarrow (L)L'$ | | $L \longrightarrow aL'$ | | |
| L' | | $L' \longrightarrow \epsilon$ | | $L' \longrightarrow , SL'$ | |

(8) 1) Function Preserving Transformations.

There are a number of ways in which a compiler can improve a program without changing the function it computes. The transformations common sub expression elimination, copy propagation, dead-code elimination and constant folding are common examples of such function preserving transformations.

2) Operator precedance relation.

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow id$

$\rightarrow E \rightarrow E+E \mid E*E \mid id$

Grammar is not operator precedance

$S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b.$

Converted grammar

$S \rightarrow SbSbS \mid SbS \mid a$
$A \rightarrow bSb \mid b.$

Relation Table for $E \rightarrow E+E \mid E*E \mid id$

| | id | + | * | $ |
|---|---|---|---|---|
| id | | $\cdot>$ | $\cdot>$ | $\cdot>$ |
| + | $<\cdot$ | $\cdot>$ | $<\cdot$ | $\cdot>$ |
| * | $<\cdot$ | $\cdot>$ | $\cdot>$ | $\cdot>$ |
| $ | $<\cdot$ | $<\cdot$ | $<\cdot$ | |

(11) 1) Role of finite automata in lexical analysis.
The finite automata is the combination of five tuples focusing on states and transition through input symbols. In the design of a compiler, it used in lexical analysis to produce tokens in the form of identifiers, keywords & constants from the input program.

2) If your program has a large collection of reversed words it is more efficient to let us simply match a string and determine in your own code if it is a variable or reversed word

3) Input Buffer in lexical analysis. The lexical analyser scans the input from left to right one character at a time. It uses a pointer begin and forward to keep track of the pointer of the input scanned

4) First(S) = { a, ∈, c, d, e, f }
First(P) = { a, ∈, c, f }
First(Q) = { c, d, ∈ }
First(R) = { e, f }

Follow(S) = { $ }
Follow(P) = { c, d, $, e, f }
Follow(Q) = { e, f }
Follow(R) = { $, b }

(16) 1) Model of Activation record.

A general activation record consists of the following things :-

(i) Local variables : hold the data that is local to the execution

(ii) Temporary values : stores the values that arise in the evaluation of an expression

(iii) Machine Status : holds the information about status of machine just before function call.

(iv) Access link : non-local data in other activation records

(v) Control link : points to activation record of caller

(vi) Return Value : used by the called procedure to return a value to calling procedure

(vii) Actual Parameters

LALR(1) Parser

$S \rightarrow iCtS \mid CtSeS \mid a$

$C \rightarrow b.$

Start Symbol S.

$S' \rightarrow S$

$S \rightarrow iCtS$

$S \rightarrow CtSeS$

$S \rightarrow a$

$C \rightarrow b$

$I_0$ (Augmented Grammar)

$S' \rightarrow .S$ , $

$S \rightarrow .iCtS$ , $

$S \rightarrow .CtSeS$ , $

$S \rightarrow .a$ , $

$C \rightarrow .B$ , t

$I_1$ goto (0, S)

$S' \rightarrow S.$ , $

$I_2$ goto (0, i)

$S \rightarrow i.CtS$ , $/e

$C \rightarrow .B$ , t

$I_3$ goto (0, C)

$S \rightarrow C.tSeS$ , $/e

$I_4$ goto (0, a)

$S \rightarrow a.$ , $/e

$I_5$ goto (0, B)

$C \rightarrow B.$ , t

$I_6$ goto (2, C)

$S \rightarrow iC.tS$ , $/e

$I_7$ goto (3, t)

$S \rightarrow Ct.SeS$ , $/e

$S \rightarrow .iCtS$ , e

$S \rightarrow .CtSeS$ , e

$S \rightarrow .a$ , e

$C \rightarrow .B$ , t

$I_8$ goto (6, t)

$S \rightarrow iCt.S$ , $/e

$I_9$ goto (7, S)

$S \rightarrow$ Ct s . eS , $/e

$I_{10}$ goto (8, s)

$S \rightarrow$ i Cts , $/e

$I_{11}$ goto (9, e)

$S \rightarrow$ Cts e.S , $/e

$S \rightarrow$ . iCts , $/e

$S \rightarrow$ . Ctses , $/e

$S \rightarrow$ . a , $/e

$C \rightarrow$ . B , t

$I_{12}$ goto (11, s)

$S \rightarrow$ Ctses . , $/e

(14) ▶ CLR

$S \rightarrow AA$

$A \rightarrow Aa \mid b$

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow Aa$

$A \rightarrow b$

**$I_0$ (Augmented Gramma)**

$S' \rightarrow .S$ , $\$$

$S \rightarrow .AA$ , $\$$

$A \rightarrow .Aa$ , $b/a$

$A \rightarrow .b$ , $b/a$

**$I_1$ goto (0, S)**

$S' \rightarrow S.$ , $\$$

**$I_2$ goto (0, A)**

$S \rightarrow A.A$ , $\$$

$A \rightarrow A.a$ , $a/b$

$A \rightarrow .Aa$ , $\$/a$

$A \rightarrow .b$ , $\$/a$

**$I_3$ goto (0, b)**

$A \rightarrow b.$ , $b/a$

**$I_4$ goto (2, A)**

$S \rightarrow AA.$ , $\$$

$A \rightarrow A.a$ , $\$/a$

**$I_5$ goto (2, a)**

$A \rightarrow Aa.$ , $b/a$

**$I_6$ goto (2, b)**

$A \rightarrow b.$ , $\$/a$

**$I_7$ goto (4, a)**

$A \rightarrow Aa.$ , $\$/a$

First (S') = {L}

First (S) = {b}

First (S) = {L}

CLR Table

| State | ACTION | | | GOTO | | |
|---|---|---|---|---|---|---|
| | a | b | $ | S' | S | A |
| 0 | | s3 | | | 1 | 2 |
| 1 | | | acc | | | |
| 2 | s5 | s6 | | | | 4 |
| 3 | r3 | r3 | | | | |
| 4 | s7 | | r1 | | | |
| 5 | r2 | r2 | | | | |
| 6 | r3 | | r3 | | | |
| 7 | r2 | | r2 | | | |

2) → Code Motion : It means that the code is moved out of the loop as it won't have any difference if it is performed inside the loop repeatedly.

→ Copy Propagation : Process of replacing the occurences of targets of direct assignments with their values

→ Dead Code Elimination : It is a compiler optimization to remove code which dosnot affect the programs results.

(21) 1) Viable Prefixes.

Viable Prefix is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

This clearly means that a viable prefix has a handle at its rightmost end.

Not all prefixes of right sentential form can appear on the stack of a shift reduce parser.

2) FIRST & FOLLOW

$S \rightarrow A$

$A \rightarrow aBcB \mid B \mid D$

$B \rightarrow dE$

$D \rightarrow dE$

$E \rightarrow FcA \mid FcC$

$C \rightarrow c \mid \epsilon$

$F \rightarrow b$

| Non-Terminal | First | Follow |
|---|---|---|
| S | {a, d} | {$} |
| A | {a, d} | {$, c} |
| B | {d} | {c, $} |
| D | {d} | {$, c} |
| E | {b} | {c, $} |
| C | {c, ε} | {c, $} |
| F | {b} | {c} |

(18) 1) Criteria for achieving machine dependent optimisation:

It 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 ~~reference~~ rather than relative reference.

2) Canonical parser diagram.

$S \rightarrow CC$

$C \rightarrow cC \mid d$

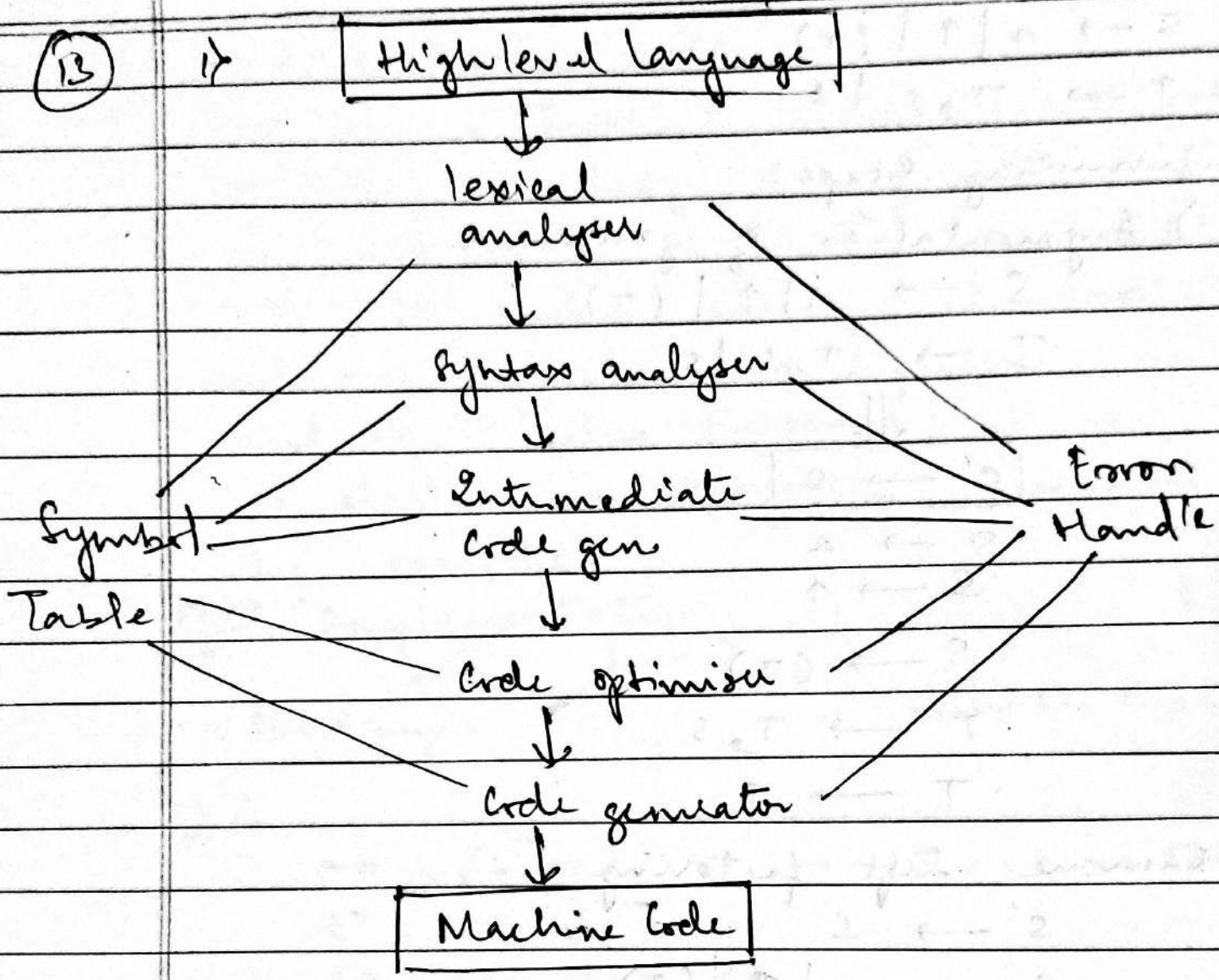
$S' \rightarrow .S$

$S \rightarrow .CC$

$C \rightarrow .cC \mid .d$

| | |
|---|---|
| $\underline{I_0}$ | $\underline{I_3}$ |
| $S' \rightarrow .S$ | $C \rightarrow c.C$ |
| $S \rightarrow .CC$ | $C \rightarrow .cC \mid .d$ |
| $C \rightarrow .cC$ | |
| $C \rightarrow .d$ | $\underline{I_4}$ |
| | $C \rightarrow d.$ |
| $\underline{I_1}$ | |
| $S' \rightarrow S.$ | $\underline{I_5}$ |
| | $S \rightarrow C(.$ |
| $\underline{I_2}$ | |
| $S \rightarrow C.C$ | $\underline{I_6}$ |
| $C \rightarrow .cC \mid .d$ | $C \rightarrow cC.$ |

(13) 1)




a = b + c * 50.

↓

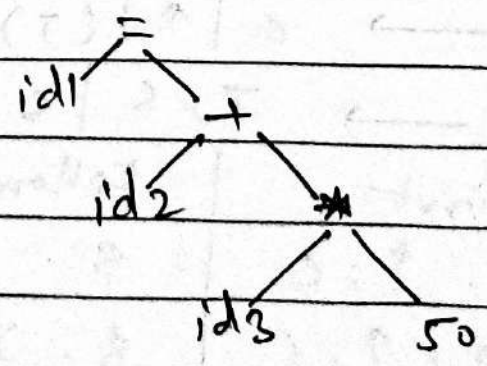
Lexical Analyser

id1 = id2 + id3 * 50

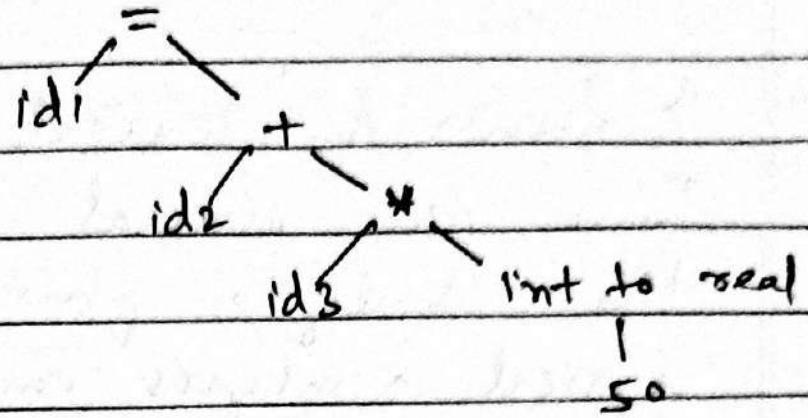
↓

Syntax Analysis

| Symbol Table | |
|---|---|
| a | id1 |
| b | id2 |
| c | id3 |



Semantic Analysis




Intermediate Code Generator

temp1 = int to real(50)

temp2 = id3 * temp1

temp3 = id2 + temp2

id1 = temp3

Code optimise

temp1 = id3 * 50.0

id1 = id2 + temp1

Code generator

```
MOV   id3, R2
MULF  #50.0, R2
MOVF  id2, R1
ADDF  R2, R1
```

# LR(1) Parser.

$S \longrightarrow x \mid Ay$

$B \longrightarrow \epsilon \mid z$

$A \longrightarrow Bx$

$S' \longrightarrow S$

$S \longrightarrow x$

$S \longrightarrow Ay$

$B \longrightarrow \epsilon$

$B \longrightarrow z$

$A \longrightarrow Bx$

$I_0$ (Augmented Grammar)

$S' \rightarrow . S \;,\; \$$

$S \rightarrow . x \;,\; \$$

$S \rightarrow . Ay \;,\; \$$

$A \rightarrow . Bx \;,\; y$

$B \rightarrow . \;,\; x$

$B \rightarrow . z \;,\; x$

$I_1$ goto $(0, S)$

$S' \rightarrow S. \;,\; \$$

$I_2$ goto $(0, x)$

$S \rightarrow x. \;,\; \$$

$I_3$ goto $(0, A)$

$S \rightarrow A.y \;,\; \$$

$I_4$ goto $(0, B)$

$A \rightarrow B.x \;,\; y$

$I_5$ goto $(0, z)$

$B \rightarrow z. \;,\; x$

$I_6$ goto $(3, y)$

$S \rightarrow Ay. \;,\; \$$

$I_7$ goto $(4, x)$

$A \rightarrow Bx. \;,\; y$

First (S') = { x, ϵ, z, y }

First (S) = { x, ϵ, z, y }

First (B) = { ϵ, z }

First (A) = { ϵ, z, x }

LR(1) Table

| | ACTION | | | | GOTO | | | |
|---|---|---|---|---|---|---|---|---|
| State | x | y | z | $ | S' | S | B | A |
| 0 | s2/r3 | | s5 | | | 1 | 4 | 3 |
| 1 | | | | acc | | | | |
| 2 | | | | $r_1$ | | | | |
| 3 | | s6 | | | | | | |
| 4 | s7 | | | | | | | |
| 5 | $r_4$ | | | | | | | |
| 6 | | | | $r_2$ | | | | |
| 7 | | $r_5$ | | | | | | |

(4) a)

a) Common sub expression elimination :

CSE is a compiler optimisation that searches for instances of identical expressions, ie, they evaluate to the same value and analyse if its worth while replacing with a single variable

Eg code :

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

Transformed code :

```
temp = b * c;
a = temp + g;
d = temp * e;
```

b) Copy propagation :

Process of replacing the occurrences of targets of direct assignments with their values.

Code :

```
y = x;
z = 3 + y;
```

Copy Propagation :

```
z = 3 + x;
```

c) Dead code elimination :

Removes code which does not affect the program results.

| Code : | Removing dead code : - |
| --- | --- |
| int global ;<br>void f() {<br>int i ;<br>i = 1 ;<br>global = 1 ;<br>global = 2 ;<br>return ;<br>global = 3 ; } | int global ;<br>void f() {<br>global = 2 ;<br>return ; } |

d) Code Motion :

Refers that the code is moved out of the loop as it won't have any difference if it is performed inside the loop repeatedly or outside the loop once.

```
for (int x=0 ; x < string.length() ; x++)
{ // Other code here
}
```

2) Eliminating left recursion, etc. :-

$E \rightarrow E+T \mid T$

$T \rightarrow id \mid id() \mid id(x)$

$x \rightarrow E, E \mid \epsilon$

Left recursion elimination :

$E \rightarrow E+T \mid T \Longrightarrow E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

Left factoring

$T \rightarrow id \mid id() \mid id(x)$

$\Downarrow$

$T \rightarrow idT'$

$T' \rightarrow \epsilon \mid () \mid (x)$

$x \rightarrow E, E \mid \epsilon$

$\Downarrow$

$x \rightarrow Ex'$

$x' \rightarrow , E \mid \epsilon$

Lang!-

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow idT'$

$T' \rightarrow \epsilon \mid () \mid (x)$

$x \rightarrow Ex'$

$x' \rightarrow , E \mid \epsilon$

| | First | Follow |
|---|---|---|
| E | id | ), $, ) |
| E' | +, ϵ | ), $, ) |
| T | id | +, ), $, ) |
| T' | ϵ, ( | +, ), $, ) |
| x | id | ) |
| x' | , , ϵ | ) |