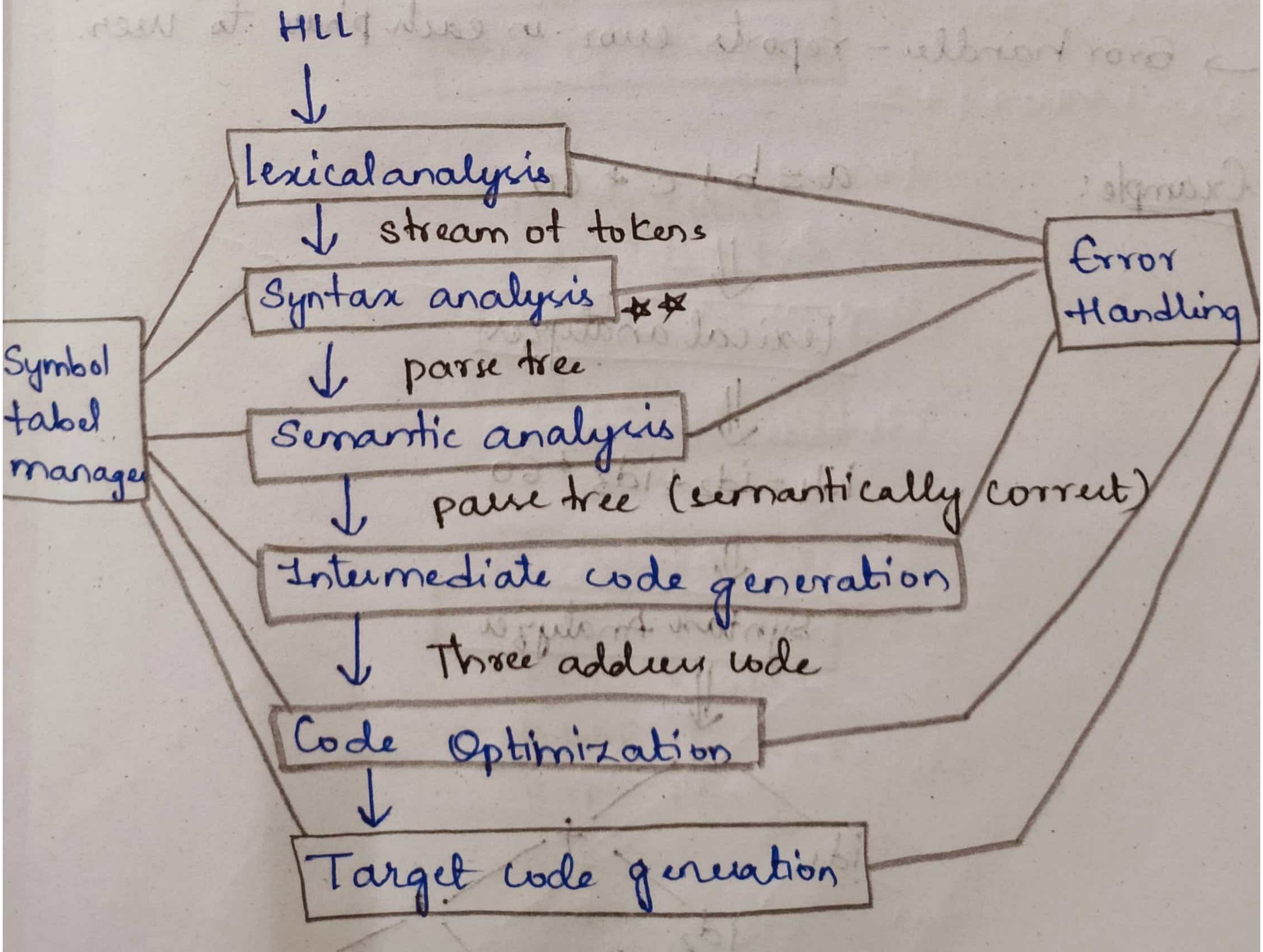
4/3/2024

Unit-2

Language preprocessing system

HLL
↓
Pre processor
↓ pure HLL
Compiler
↓ Assembly Language
Assembler
↓ Machine code (relocatable)
Loader/Linker
↓
Executable code / Absolute machine code

Phases of Compiler



→ Pre processor removes #include<stdio.h> (File inclusion)
#define (macro expansion)

→ Address present in main memory (physical address)
Address generated by CPU (logical address)

→ Software developed for user need (Application Software)
for system need (System software)

→ Phases (6) – first 4 (frontend/analysis)
next 2 (backend/synthesis)

→ Popular Intermediate Code (Three address Code)
↓
max no. of addresses in any instruction '4'

→ Symbol table – a datastructure that stores information about each phase.

→ Error handler – reports error in each phase to user.

<u>Example:</u>

$$a = b + c * 60$$

⇓

$$\boxed{\text{Lexical analyzer}}$$

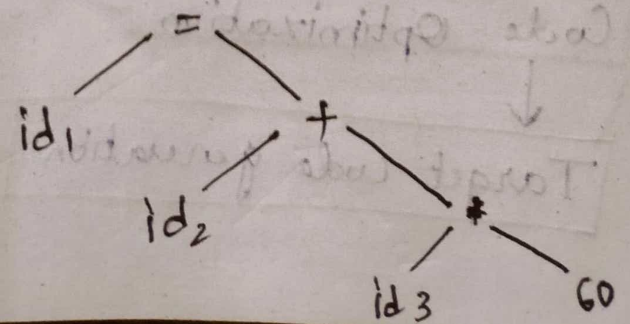
⇓

$$id_1 = id_2 + id_3 * 60$$

⇓

$$\boxed{\text{Syntax Analyzer}}$$

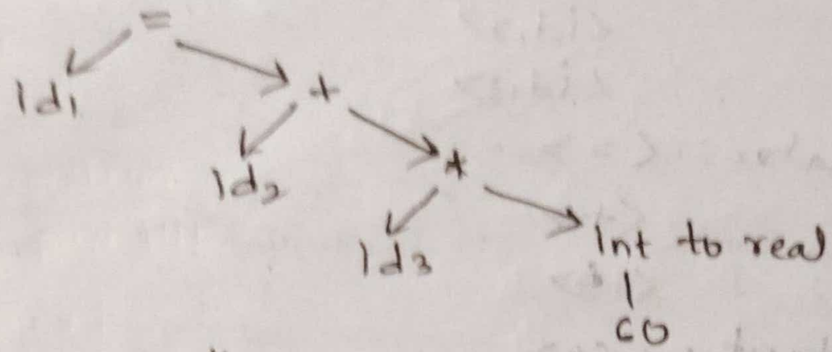
⇓



⇓

Semantic Analyzer

⇓



⇓

Intermediate Code generation

⇓

t_1 = int to real(60)
t_2 = $id_3 * t_1$
t_3 = $id_2 + t_2$
id_1 = t_3

⇓

Code Optimization

⇓

t_1 = $id_3 * 60.0$
id_1 = $id_2 + t_1$

⇓

Code Generation

⇓

```
LDF   R2, id3
MULF  R2, R3, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

Token:

for identifier : <Token name, attribute value>

<id,1>

<id,2>

<id,3>

for operator : <=>

<+>

<*>

for constant : <60>

In three address code :

1) RHS should have atmost one operator

LDF : loading floating point variable

STF : storing floating point variable

MULF: multiply

ADDF : Add

1) $i = i * 70 + j + 2$

$i = i * 70 + j + 2$

⇓

$\boxed{\text{lexical analyzer}}$

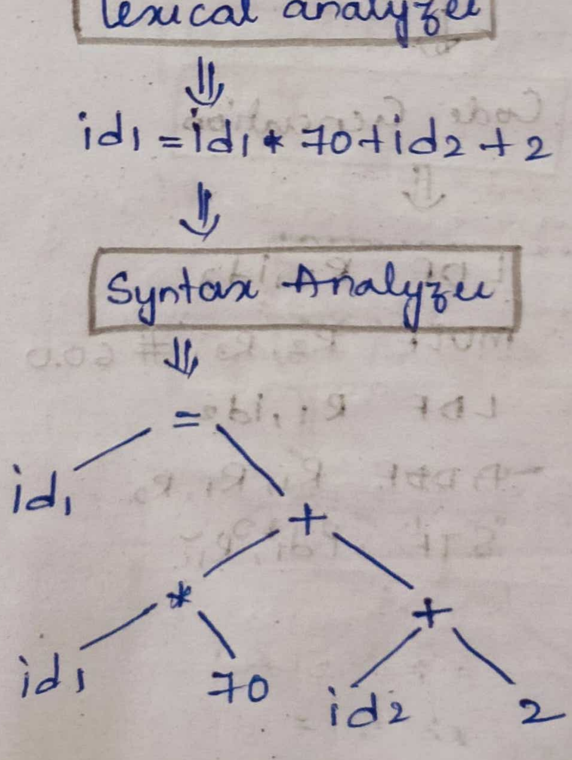
⇓

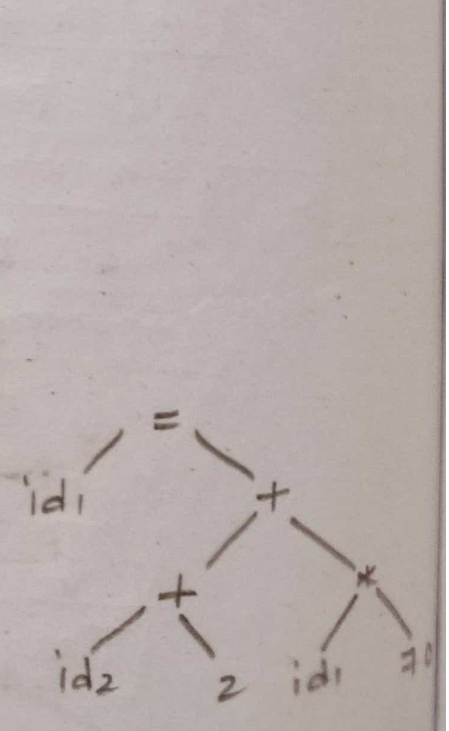
$id_1 = id_1 * 70 + id_2 + 2$

⇓

$\boxed{\text{Syntax Analyzer}}$

⇓

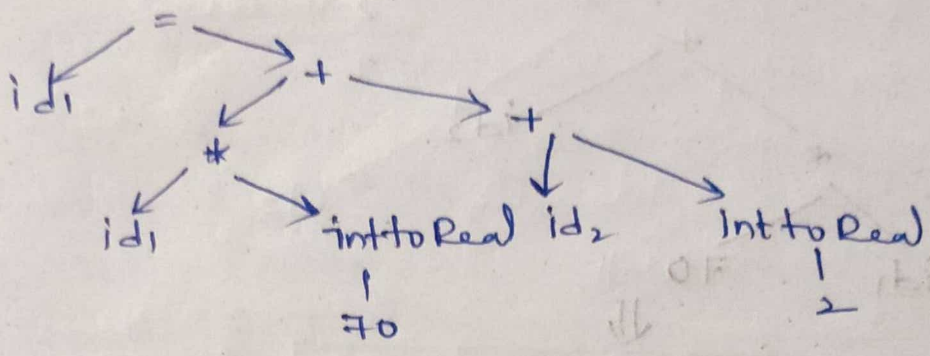




⇓

Semantic Analyzer

⇓



⇓

Intermediate code generation

⇓

$t_1 = inttoReal(70)$

$t_2 = id_1 * t_1$

$t_3 = inttoReal(2)$

$t_4 = id_2 + t_3$

$t_5 = t_2 + t_3$

$id_1 = t_5$

⇓

Code optimization

⇓

$t_1 = id_1 * 70.0$

$t_2 = id_2 + 2.0$

$id_1 = t_1 + t_2$

⇓

Code Generation

⇓

LDF R1, id1

MULF R1, R1, # 70.0

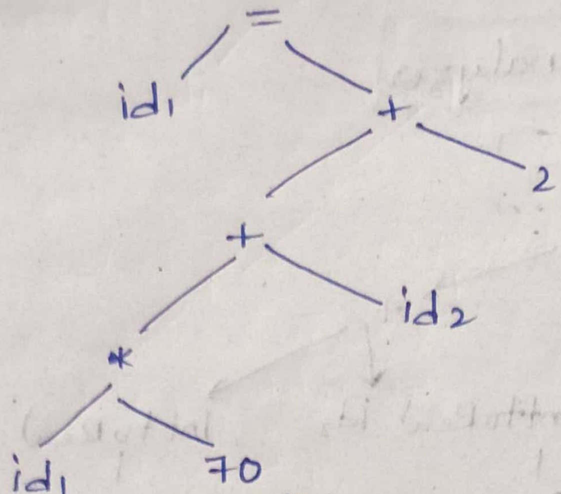
LDF R2, id2

ADDF R2, R2, # 2.0

ADDF R2, R1, R2

STF id1, R2

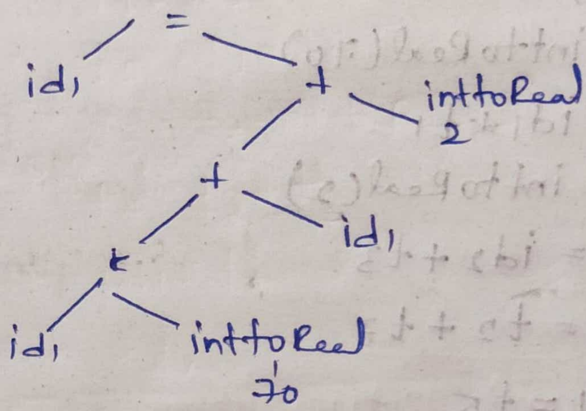
Sol)



⇓

Semantic Analyzer

⇓



⇓

Intermediate code generation

⇓

$t_1 = \text{inttoReal}(70)$

$t_2 = id_1 * t_1$

$t_3 = t_2 + id_2$

$t_4 = \text{inttoreal}(2)$

$t_5 = t_3 + t_4$

$id_1 = t_5$

⇓

Code Optimization

⇓

$t_1 = id_1 * 70.0$

$t_2 = t_1 + id_2$

$t_3\ \cancel{id_1} = t_2 + 2.0$

$id_1 = t_3$

⇓

$\boxed{\text{Code generation}}$

⇓

```
LDF   R1, id1
MULF  R1, R1, #70.0
LDF   R2, id2
ADDF  R1, R1, R2
ADDF  R1, R1, #2.0
STF   id1, R1
```

2) $a = (b+c) * (b+c) + 2$

$a = (b+c) * (b+c) + 2$

⇓

$\boxed{\text{lexical Analyzer}}$ (Scanner)

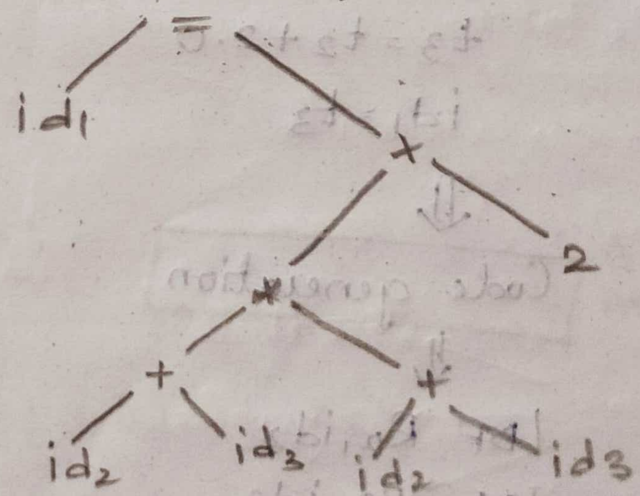
⇓

$id_1 = (id_2 + id_3) * (id_2 + id_3) + 2$

⇓

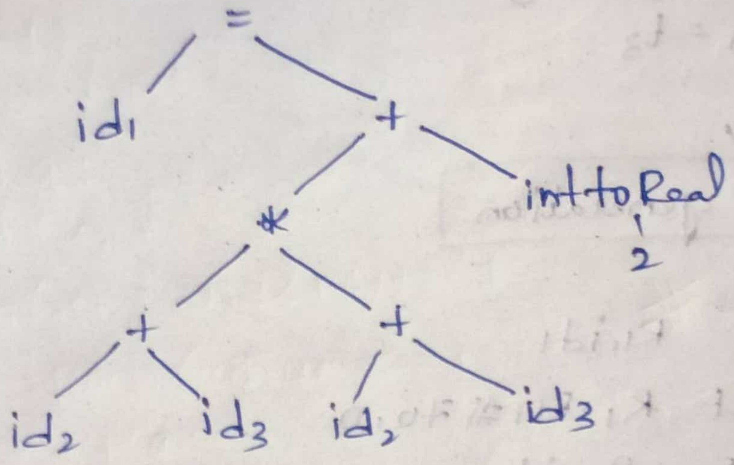
$\boxed{\text{Syntax Analyzer}}$ (Parser)

⇓



⇓

$\boxed{\text{semantic Analyzer}}$



⇓

Intermediate Code generation

⇓

$t_1 = id_2 + id_3$

$t_2 = t_1 * t_1$

$t_3 = \cancel{t_2} \cdot \text{int-to Real}(2)$

$t_4 = t_2 + t_3$

$id_1 = t_4$

⇓

Code Optimization

⇓

$t_1 = id_2 + id_3$

$t_2 = t_1 * t_1$

$t_3 = t_2 + 2.0$

$id_1 = t_3$

⇓

Code generation

⇓

LDF R_2, id_2

LDF R_3, id_3

ADDF R_2, R_2, R_3

MULF R_2, R_2, R_2

ADDF R2, R2, #2.0

~~LDF R1, id1~~

STF id1, R2

11/03/2024

4, 6, 9, 13, 14, 16

4) Number of tokens in the following statement

printf("i = %d, &i = %x", i, &i); # 10 tokens

6) int strange(int x)

```
{
    if(x <= 0) return 0;
    if(x % 2 != 0) return x-1;
    return 1 + strange(x-1);
}
```

42 tokens

9) main()

```
{
    char ch = 'A';
    int x, y;
    x = y = 20;
    x++;
    printf("%d %d", x, y);
}
```

33 tokens

13) main()

```
{
    int *a, b;
    b = 10;
    a = &b;
    printf("%d %d", b, *a);
    b = */* pointer */b;
}
```

~~40~~ 35 tokens

14)
```
main()
{
    int a;
    int *a;
    for(int i=0; i<n; i++){
        printf("True");
        ++*a;
        a=a+a;
    }
}
```
42 tokens

16)
```
int a=10;
float b=20.5;
printf("c=%d,d=%.", ++a,b++);
```
21 tokens

2)
```
switch(input value)
{
    Case 1: b=c*d; break;
    default: b=b++; break;
}
```
27 tokens

Responsibilities of Lexical analyzer:

1) Generate tokens
2) Eliminates comments
3) Eliminates white spaces

Tokens → identifiers
Tokens → Keywords
Tokens → Operators
Tokens → Special symbol

<u>Unit-1</u>

Pumping lemma for regular languages:

It is used to prove some of the languages are not regular.

- > Finite language → Regular
- > Infinite language — Regular / Not Regular

* $L = \{a^n b^n \mid n \geq 0\}$
  $L = \{ww^R \mid w \in (a+b)^*\}$
  $L = \{a^p \mid p \text{ is a prime number}\}$

1) $L = \{a^n b^n / n \geq 0\}$

Assume L is a regular language

Take some n-value (pumping length)

aaaaaabbbbbb

$n = 6$

Divide given string into 3 parts xyz

<u>Case-1</u>:

$$\underbrace{aaaa}_{x} \mid \underbrace{aab}_{y} \mid \underbrace{bbbbb}_{z}$$

Pump xy^iz

let $i = 2$

$\Rightarrow$ $\underbrace{aaaa\,aab\,aab\,bbbbb}_{\text{It is not present in Language}}$

<u>Case 2</u>: aaaa | aabb | bbbb

$\underbrace{aaaa\,aabbaabbbbb}_{\text{not present}}$

By contradiction, L is not regular Language

2) $L = \{ww^R / w \in (a+b)^+\}$

Let L is a regular language

18/3/24

Context free Grammar (CFG):

$G = (V, T, P, S)$

$\rightarrow A \rightarrow \alpha$

$\alpha \in (V \cup T)^*$

V: Variables

T: Terminals

P: productions

S: start symbol

1) $E \rightarrow E+E \,/\, E*E \,/\, id$

$V = \{E\}$

$T = \{+, *, id\}$

Derivations → LMD, RMD

Derivation / Parse tree

LMD: $id+id*id$

$E \Rightarrow E+E$

$\Rightarrow id+E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$