8th February

Language Processors

- Converts given language to target language.

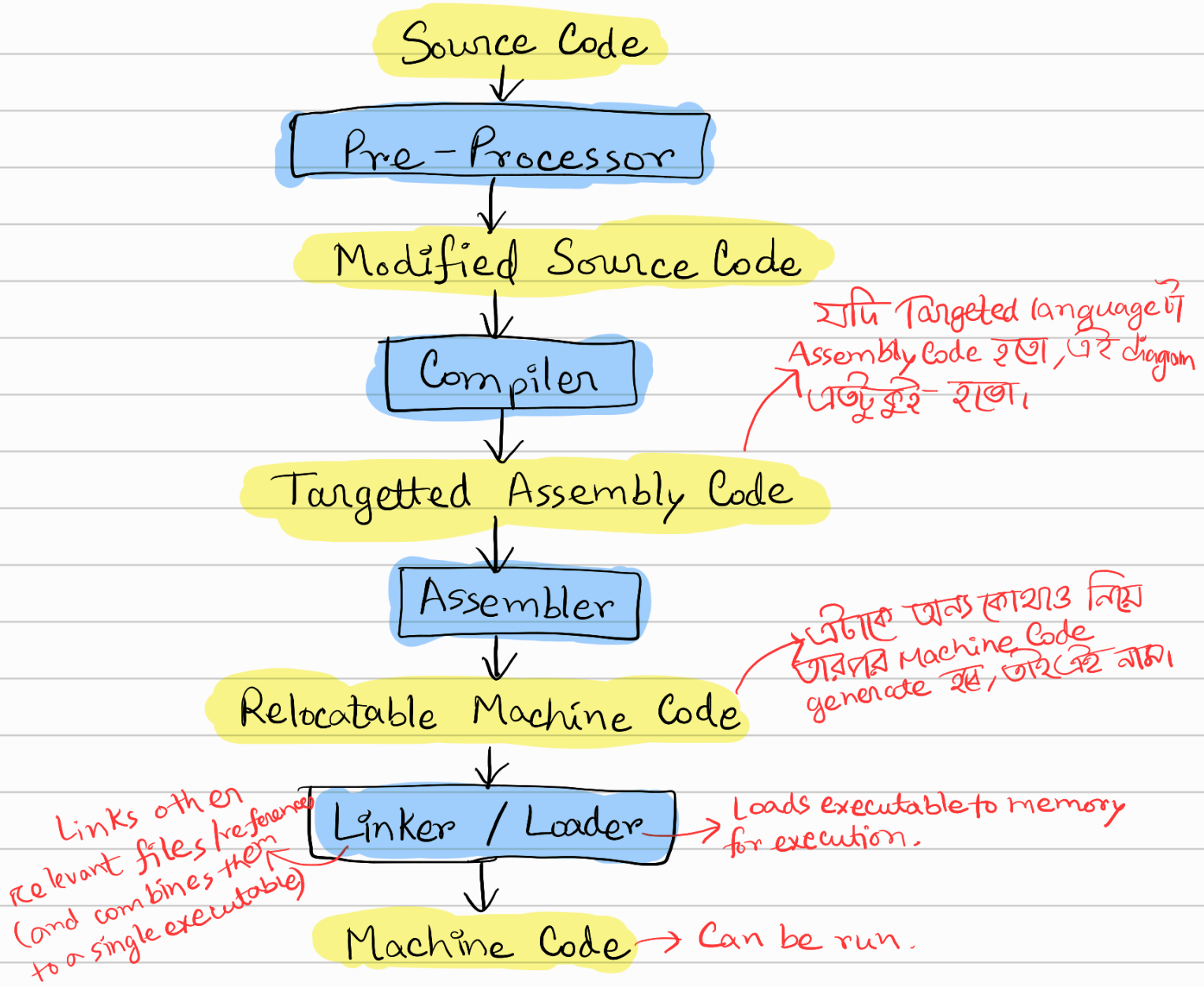
→ Assembler
→ Compiler → পুরোটার compilation শেষে হলে error messages দেয়
→ Interpreter → Runs line by line

Possible Questions:

Q) Diff. betⁿ Compiler and Interpreters

Language Processing System:

The Compilation Process Diagram:



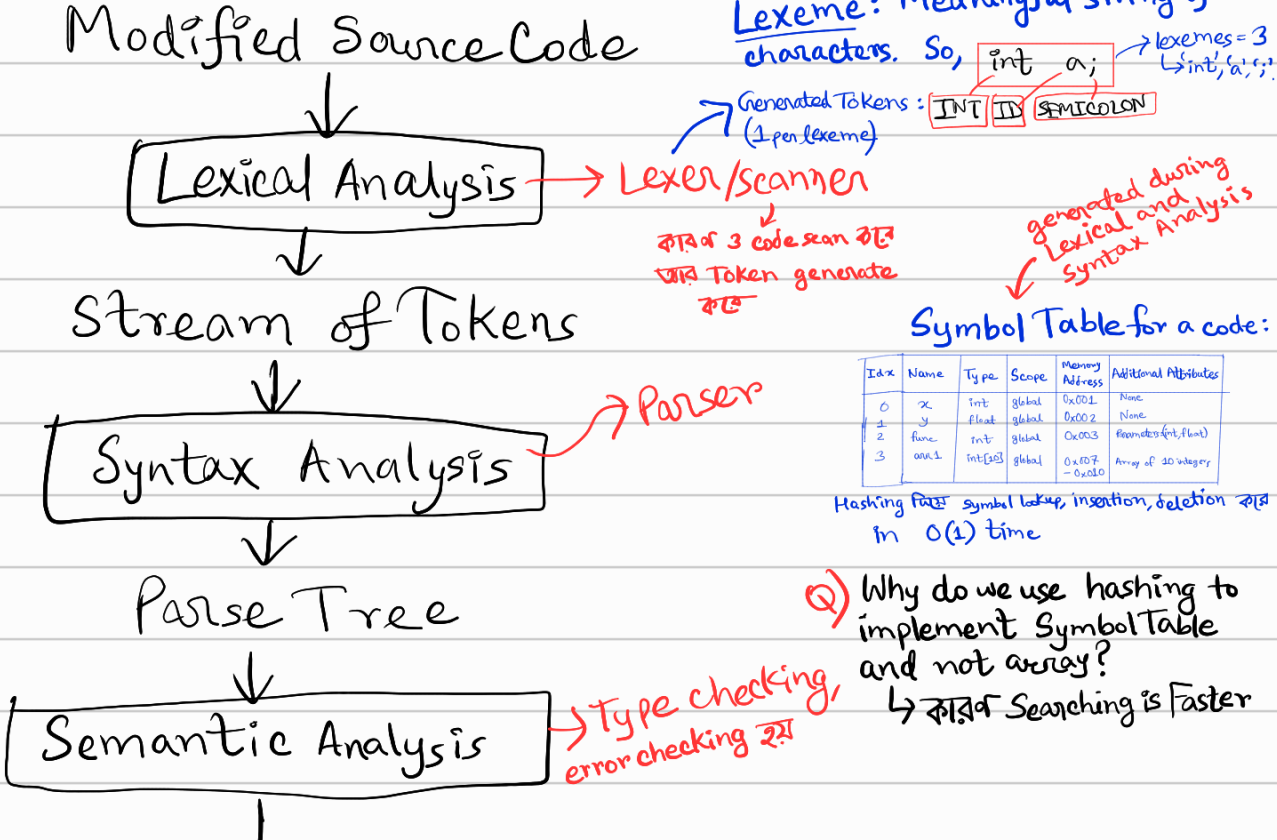
Pre-processing steps:

1) Removing hashtags. → Hashtags like #include <scope_table.h>
2) Adding respective files. → Getting scope_table.h file and adding it.
3) Macro expansion → Replacing constants with their actual values. eg: pi becoming 3.14.
4) Operator conversion → Replacing operator short forms, eg: i++ becomes i=i+1

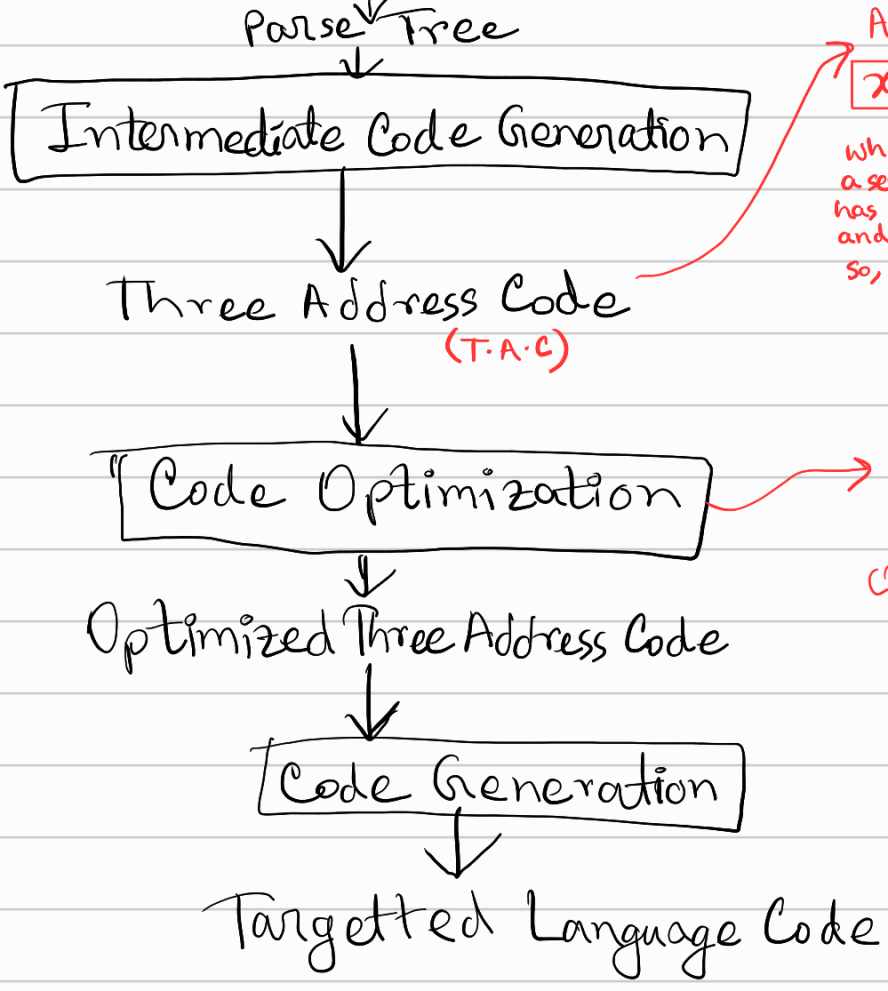
Compiler steps:

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code generation
5) Code Optimization (optional) } Code Synthesis snap
6) Code Generation

Compiler Steps



Q) Why do we use hashing to implement SymbolTable and not array?
↳ কারণ Searching is Faster



A form of code of the format

$x = y \text{ op } z$,

(op = operation)

where complex statements are broken into a series of simpler ones where each statement has at most 3 operands, simplifying analysis and optimization phases.

So, $a = b + c * d$ becomes:

- $t1 = c * d$
- $t2 = b + t1$
- $a = t2$

Source Code:

$z = a + b + c$

Three Addr. Code:

$t1 = a + b$
$t2 = t1 + c$
$z = t2$

't' = temporary

after optimization:

$t1 = a + b$
$z = t1 + c$

Every phase has a error handler connected to it.

13th Feb

Syntax Error (found during Syntax Analysis)

→ missing or misplaced punctuations
→ incorrect use of keywords (both lexical and syntax analysis phase এ ধরা পড়তে পারে)
→ mismatched brackets
→ incorrectly structured statements

fi(a==b)
Q) lexical phase এ ধরা না পড়লেও syntax এ পড়বে, why?
↳ Hint: Has something to do with ID generation and Token generation

Semantic Error

→ Using variables which have not been declared

→ Type mismatch

→ Misuse of operators

→ Logical Errors

Regex important for midterm

Pattern match করলে Token generate হয়

প্রতিটি lexeme এর জন্য একটা token generate হবে।

Token Generation :

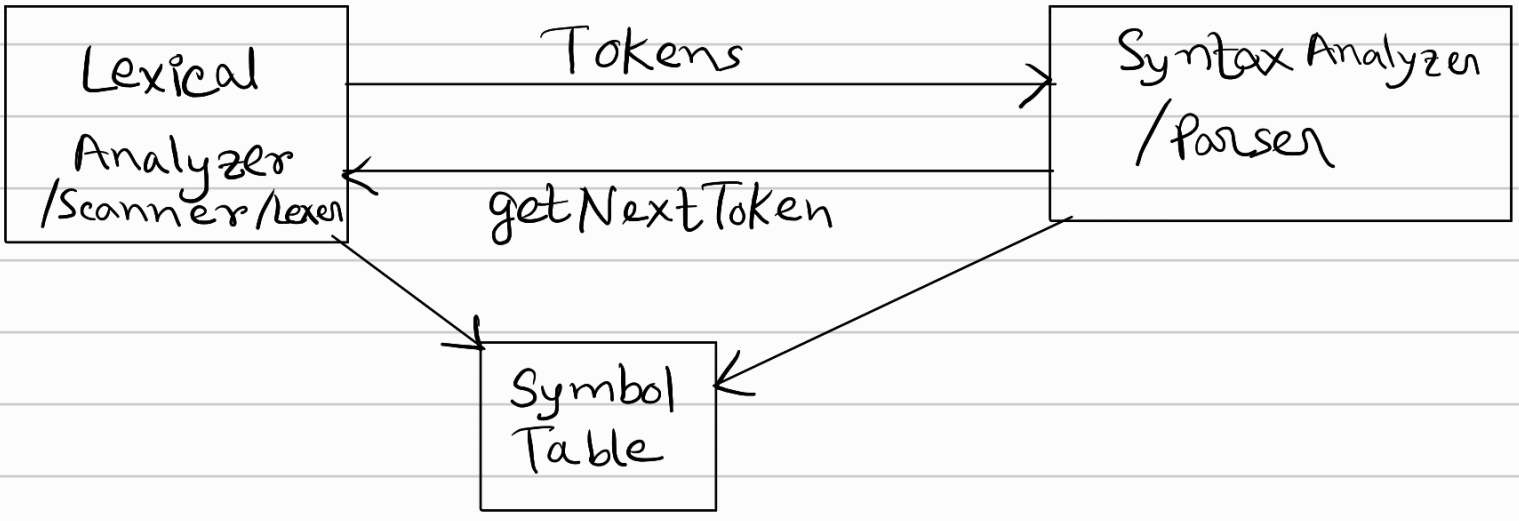
```
int  sum  (  x  ,  y  )  {
 1    2   3  4  5  6  7  8

 return   x  +  y  ;
   9      10 11 12 13

 }
 14
```

So, No of lexemes = 14
∴ No of tokens = 14

The Role of Lexical Analyzer:



ধরো source code:

```
int a;
```

Tokens: INT ID SEMICOLON

এগুলা আগে lexer এ যাবে, পরে Parser এ যায়।
Lexer has patterns it matches code against, to generate Tokens
Parser এর কাছে (regex) grammer দেওয়া থাকে, against which
3 check করে for syntax errors.

Summary

→ lexical Analyzer reads the input characters from the source program

→ Lexical Analyzer then groups them into lexemes

→ Lexical Analyzer then produces tokens for the lexemes

→ Stream of tokens are then sent to Syntax Analyzer/Parser

→ When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter the lexeme information on symbol table.

→ getNextToken: This function acts as the interface between parser and lexer. Parser requests the next token from the lexer by calling it. The lexer scans the source code and generates tokens.

So, flow টা হচ্ছে:
Lexemes → Patterns (regex) → Tokens

Panic mode recovery

```
fi+(a==b)
```

Token generate হচ্ছে না,
৩ একটা একটা করে character
বাদ দেয় আর pattern match করে token generate করার চেষ্টা করতে থাকে (left to right)
তারপর token generate করতে পারলে sends that to Parser.

কোনো কারণে lexer token generate করতে না পারলে সে alternate recovery methods use করে, mainly Panic Mode recovery

So, করতে if(a==b) দিয়ে try করে, পারবে না,
then, +(a==b) " " " , পারবে (+ কে পাঠিয়ে দেয় as ADD OPERAND)

Parts of a Token: সবচেয়ে generically token এ থাকে:

TOKEN
⟨token name, attribute value⟩

↳ Symbol table এ কত index এ এটা আছে সেটা, বা tokeninfo কোথায়

এটা basic, এটায়ও অনেক সমস্যা হতে পারে,
normally tokens are more complex.

Pattern Structure

if
{Token = IF}
Pattern to match against
Token to generate

int
{Token = INT}

{(a-b)+}
a to b
regex pattern
{Token = ID}