

# Compiler Design - Introduction

## ICOD632C - Compiler Design

S.Venkatesan

Assistant Professor  
Department of Information Technology  
Indian Institute of Information Technology, Allahabad

January 14, 2019

# Introduction

- **Translator** - Read a program in one (source) language and translate it into an equivalent program in another (target) language.
- **Error** - It should report errors during the translation process.
- **Execution** - If target program is an executable machine-language then it can be processed by taking inputs and produce outputs.
- **Interpreter** - Is another kind. Instead of producing a target program, it directly execute the operations specified in the source program on user supplied inputs.

Table: Comparison

Interpreter	Compiler
Better error diagnostic because it executes statement by statement	Machine language target produced by compiler is faster

# Java Language

- It uses both compiler and interpreter.
- First compiler - converts to byte code
- Second Interpreter - JVM interprets
- JIT (some java compilers) - It uses Just in Time compiler (byte codes to machine code before they run the intermediate program to process the input) to increase the processing speed.

# Other requirements

- Preprocessor - Collecting the separate modules in different files, apply macros
- Assembler - Producing assembly code is easier for compiler and easier to debug. Assembler produces the relocatable machine code from the assembly code.
- Linker - Relocatable machine code may have to be linked together with other relocatable object files and library files. It resolved memory addressed, where the code in one file may refer to a location in another file.
- Loader - Puts all of the executable object files into memory for execution.

# Structure

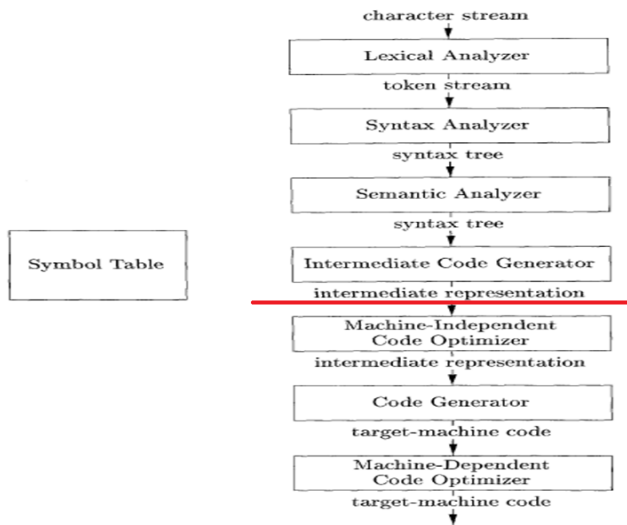


Figure: Compiler Structure

# Compiler Phase

- 1
- 2
- 3

position	...
initial	...
rate	...

SYMBOL TABLE

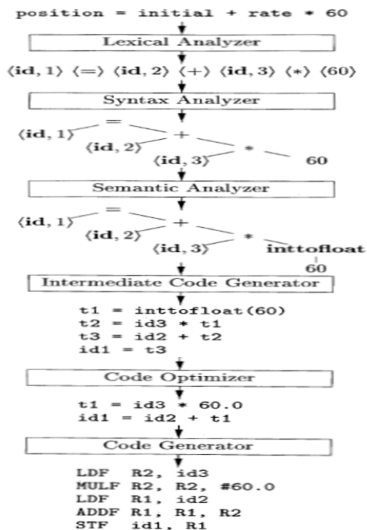


Figure: Phase

# Optimizations of Computer Architecture

- Parallelism - Instruction-level parallelism are hidden from the programmer.
- Memory Hierarchy - Using registers effectively is probably the single most important problem in optimizing a program.

# Design of Computer Architecture

- CISC - Complex Instruction Set Computer - Intel x86 architecture
- RISC - Reduced Instruction Set Computer - PowerPC, SPARC



# Program Analysis - for security vulnerabilities

- Type Checking - go beyond finding type errors.
- Bound Checking - HeartBleed attack.

# Program Language Basics

- Static and Dynamic
- Environment and states
- Explicit Access Control
- Parameter Passing
- Aliasing
- Block Structure

# Concept Map

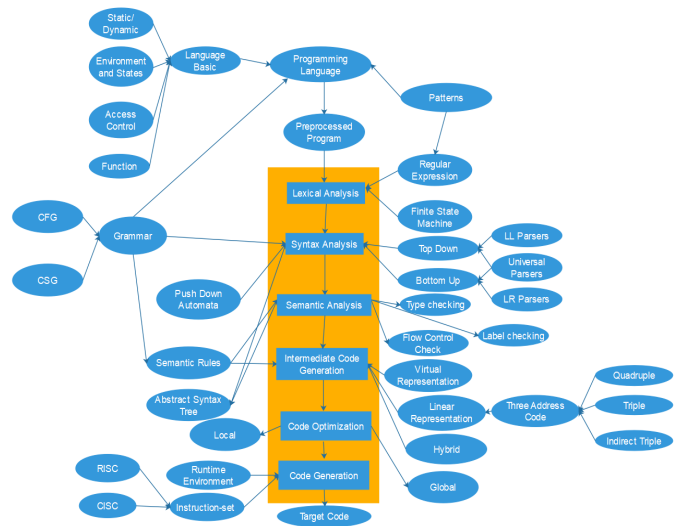


Figure: Concept Map

# Lexical Analysis

Read the input characters of the source program, group them into lexemes, and produces as output a sequence of tokens.

- Token
- Pattern
- Lexeme

Example

```
printf("Total = %d",score);
```

Id - printf & score (Lexemes)

Literal - "Total = %d" (Lexeme)

# Lexical Analysis - Process

- Scan input
- Remove white spaces
- Remove comments
- Manufacture tokens
- Generate lexical errors
- Pass token to parser

# Attribute for tokens

## Example

$$E = M * C ** 2$$

<id, pointer to symbol table E>

<assign\_op>

<id, pointer to symbol table M>

<mult\_op>

<id, pointer to symbol table C>

<exp\_op>

<number, integer value 2>

# Lexical Errors

- Ambiguity - Not able to decide because ambiguity - ex. if or fi
- It produce error, when it is not able to proceed.

## Recovery options

- Delete one character
- Insert a missing character
- Replace a character
- Transpose two adjacent characters

# Lexeme Example

Here is a photo of `<b>` my house `</b>`:

```<BR>`

See `<a>`More Pictures`</a>` if you liked that one.

Hint:

Assume HTML has well-defined tags like `<openTag>``<closeTag>`.

Ignores self-closing tags like `<br />` and `<img .. />`.

Further, ignore attributes.

Next is to have `<`, `</`, `>`, and `/>` all be separate lexical tokens.

`<literal, "Here is a photo of ">` `<openTag>` `<tagName, "b">`

`<closeTag>` `<literal, "my house">` `<openCloseTag>` `<tagName, "b">`

`<literal, ":">` `<openTag>` `<tagName, "img">` `<attributeName, "alt">`

`<equals>`



# Tricky Problem - Fortran 90

- $\text{DO } 5 \text{ I} = 1.25$
- $\text{DO } 5 \text{ I} = 1,25$

# Assignment

- Create a lexical analyser for a multi-core Architecture using the discussion given by Farhanaaz and Sanju. V [3]
- Construct the Lexical analyser tokenization using regular grammar instead of regular expression referring the discussion of Haili Luo [4]

# References

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, Compilers: Principles, Techniques and Tools, Second Edition Book, 2006.
- [2] <https://acorwin.com/2012/10/18/lexical-analysis-the-role-of-the-lexical-analyzer-section-3-1/>
- [3] Farhanaaz and Sanju. V, An Exploration on Lexical Analysis, International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), pp. 253-258, 2016.
- [4] Haili Luo, The Research of Applying Regular Grammar to Making Model for Lexical Analyzer, 2013 6th International Conference on Information Management, Innovation Management and Industrial Engineering, pp.90-92, 2013.