Compiler Design and Construction

For:
- ❖ SCII /2019 (July – Sept 2022)
- ❖ Department of Computer Science and Informatics
- ❖ School of Computing and Information Technologies
- ❖ TU-K

By: Salesio M. Kiura, PhD

1

Course Purpose and Objectives

This course introduces the learners the basic techniques that underlie the practice of Compiler Construction. It starts by formal discussions on how compilers work

At the end of this course, the student should be able to:

1. Describe the main phases of compiler
2. Illustrate the issues in compiler design
3. Explain the compiler technique
4. Design a simple compiler

Indicative Course content

- Introduction to compilers: compilers and interpreters;
- Main phases of compilers: Lexical analysis, Syntax analysis, semantic analysis, Code generation;
- Issues in compiler design: symbol tables, program compilation, loading and execution; Attribute grammars; syntax-directed translation; parsers;
- Compilation techniques: one-pass and two pass; storage allocation; object code for subscripted variables;
- A simple complete compiler: Organization, Subroutine and functions compilation, Bootstrapping techniques, multi-pass compilation;
- Optimization: techniques, local, expressions, loops and global optimization.

Ref: the course outline (delivery schedule) for more details

Schedule & Timings

Timings

- Tuesdays: 7- 9 , 9 -11
- Wednesdays: 7 -9

Blended

- Face to Face sessions
- Online Sessions

Lecturer Contacts

- Prof. **Salesio** M. Kiura, Salesio.kiura@gmail.com / salesio.kiura@tukenya.ac.ke
- Office: Room D21, Tel. Contacts: 0720 370071 / 0780 370071

Moodle Platform

URL: http://elearning.tukenya.ac.ke/

You login with the @students.tukenya.ac.ke Google email Accounts:

Navigate to School of Computing via Faculty of Applied Sciences and Technology, Select correct study course and Term

To enroll yourself, use the Key: COMPILER2019

The ICT department (Upper floor of L-Block) is provides all round support

Compilers and Interpreters

REF Book:

Compilers: Principles, Techniques, and Tools is a computer science textbook by *Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman* about compiler construction

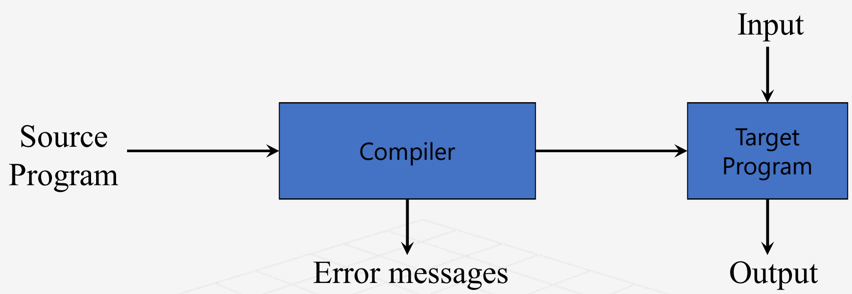
Link to download the Book:

- https://github.com/freudshow/mydoc/blob/master/docs/ALSUdragonbook.pdf

Compilers and Interpreters

a) "*Compilation*"

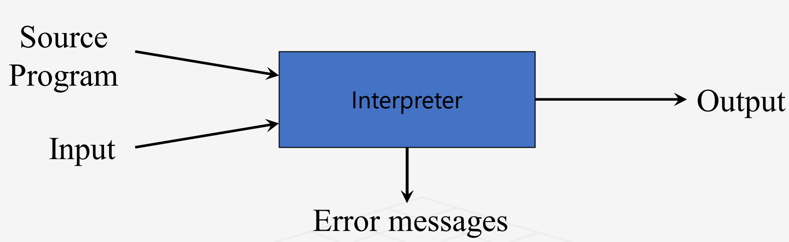
Translation of a program written in a source language into a semantically equivalent program written in a target language



Compilers and Interpreters …2

b) "*Interpretation*"

Performing the operations implied by the source program



Compiling is our focus

Compiling

To write a program takes these steps:

1. Edit the Program
2. Compile the program into Machine code files.
3. Link the Machine code files into a runnable program (also known as an exe).
4. Debug or Run the Program With some languages like Turbo Pascal and Delphi steps 2 and 3 are combined

Compile, make, build .... commands

Machine code files

Machine code files are self-contained modules of machine code that require linking together to build the final program.

The reason for having separate machine code files is efficiency; compilers only have to recompile source code that have changed.

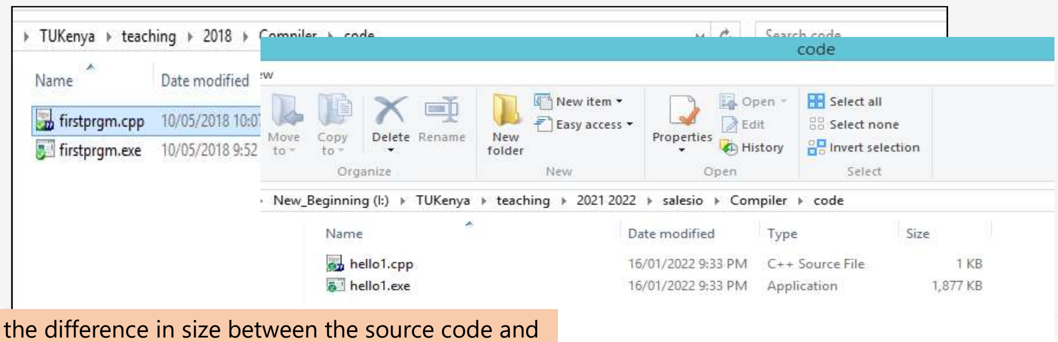
The machine code files from the unchanged modules are reused. This is known as Making the application.

If you wish to recompile and rebuild all source code then that is known as a Build.

Linking

Linking is a technically complicated process where all the function calls between different modules are hooked together, memory locations are allocated for variables and all the code is laid out in memory, then written to disk as a complete program. This is often a slower step than compiling as all the machine code files must be read into memory and linked together.

Compare and Contrast



Notice the difference in size between the source code and the compiled Application program

Java and C#

Both of these languages are semi-compiled. They generate an intermediate code that is optimized for interpretation. This intermediate language is independent of the underlying hardware and this makes it easier to port programs written in either to other processors, so long as an interpreter has been written for that hardware.

Java when compiled produces bytecode that is interpreted at runtime by a Java Virtual Machine (JVM). Many JVMs use a Just-In-Time compiler that converts bytecode to native machine code and then runs that code to increases the interpretation speed. In effect the Java source code is compiled in a two-stage process.

The Analysis-Synthesis Model of Compilation

There are two parts to compilation:

Analysis determines the operations implied by the source program which are recorded in a tree structure

Synthesis takes the tree structure and translates the operations therein into the target program

Other Tools that Use the Analysis-Synthesis Model

Editors (syntax highlighting)

Pretty printers (e.g. Doxygen)

for documentation generation

Static checkers (e.g. Lint and Splint)

for static analysis of source code: syntax errors, undeclared variables, deprecated functions,

Interpreters

E.g. …?

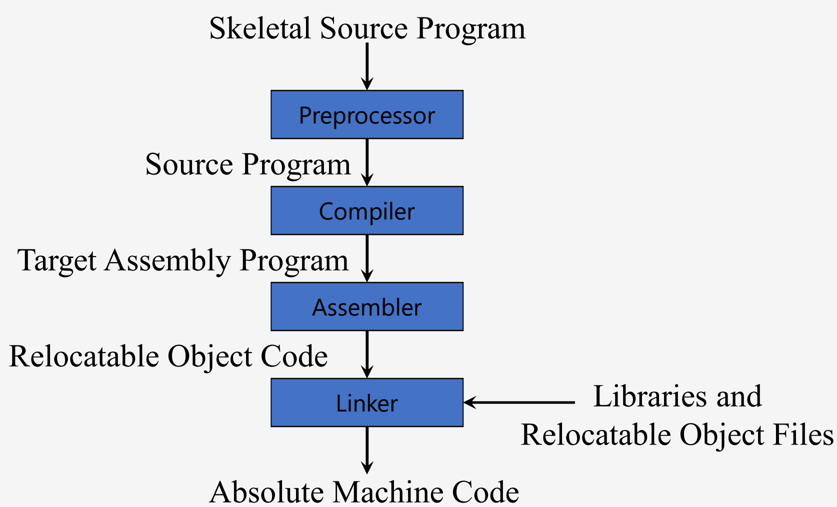
Text formatters (e.g. TeX and LaTeX)

Silicon compilers (e.g. VHDL

VHDL = VHSIC Hardware Description Language; VHSIC = very high-speed integrated circuit)

Query interpreters/compilers (Databases)

Preprocessors, Compilers, Assemblers, and Linkers



The Phases of a Compiler

Phase	Output	Sample
Programmer (source code producer)	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A', '=', 'B', '+', 'C', ';'` And *symbol table* with names

Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	`;`
		`|`
		`=`
		`/ \`
		`A   +`
		`/ \`
		`B   C`

Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	`int2fp B t1`
		`+ t1 C t2`
		`:= t2 A`

Optimizer	Three-address code, quads, or RTL	`int2fp B t1` `+ t1 #2.3 A`

Code generator	Assembly code	`MOVF #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF r2,A`

Peephole optimizer	Assembly code	`ADDF2 #2.3,r2` `MOVF r2,A`

The Grouping of Phases

Compiler *front* and *back ends*:

Front end: *analysis* (*machine independent*)

Back end: *synthesis* (*machine dependent*)

Compiler *passes:*

A collection of phases is done only once (*single pass*) or multiple times (*multi pass*)

- Single pass: usually requires everything to be defined before being used in source program
- Multi pass: compiler may have to keep entire program representation in memory

Compiler-Construction Tools

Software development tools are available to implement one or more compiler phases

- *Scanner generators*
- *Parser generators*
- *Syntax-directed translation engines*
- *Automatic code generators*
- *Data-flow engines*

Assignment

Mid Term paper

Required:

In groups, you are required to research and present a 1200 words document on the following topics.

We form the groups, then agree on how to deliver

Assign the members!

Group 1

Editors (syntax highlighting)

Pretty printers (e.g. Doxygen)

Group 2

Static checkers (e.g. Lint and Splint)

Interpreters

Group 3 (this group Must make a demo)

Text formatters (e.g. TeX and LaTeX)

Group 4

Silicon compilers (e.g. VHDL

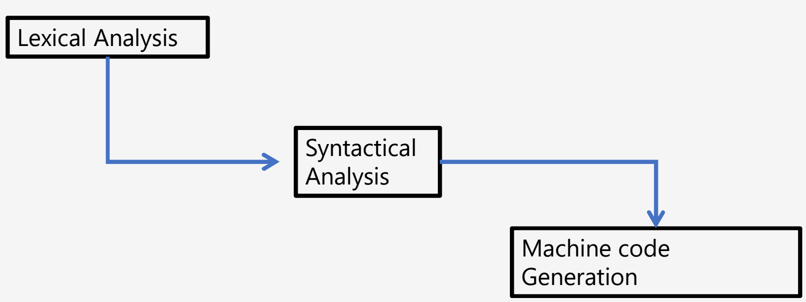
Query interpreters/compilers (Databases)

Group 5

Compiler Construction tools

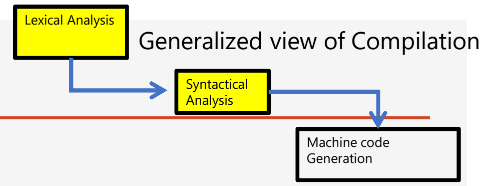
- Scanner generators
- Parser generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

Phases of a Compiler



Generalized view of Compilation

Lexical Analysis, Syntactical Analysis



Lexical Analysis

- This is the first process where the compiler reads a stream of characters (usually from a source code file) and generates a stream of lexical tokens. For example the C++ code

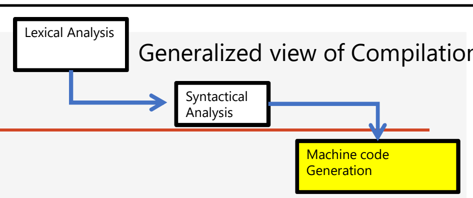
Syntactical Analysis

- The output from Lexical Analyzer goes to the Syntactical Analyzer part of the compiler.
- This uses the rules of grammar to decide whether the input is valid or not.
- E.g. given variables A and B the compiler will check:

  Declared or not declared (within a scope)

  Initialized or uninitialized

Generating Machine Code



This can be an extremely complicated process

Considerations:

The speed of the compiled executable should be as fast as possible and can vary enormously according to the quality of the generated code

How much optimization is requested:

Zero optimization - e.g. during debugging
Full optimization – for released code

Code Generation Approaches ...(1)

Some code generation approaches in modern compilers:

Instruction Pipelining

Internal caches

If all of the instructions within a loop can be held in the CPU cache then that loop will run much faster than if the CPU has to fetch instructions from main RAM. The CPU cache is a block of memory built into the CPU chip that is accessed much faster than data in the main RAM

Code Generation Approaches ...(2)

Most CPUs have a prefetch queue where the CPU reads in instructions into the cache prior to executing them. If a conditional branch happens then the CPU has to reload the queue. So code should be generated to minimize this.

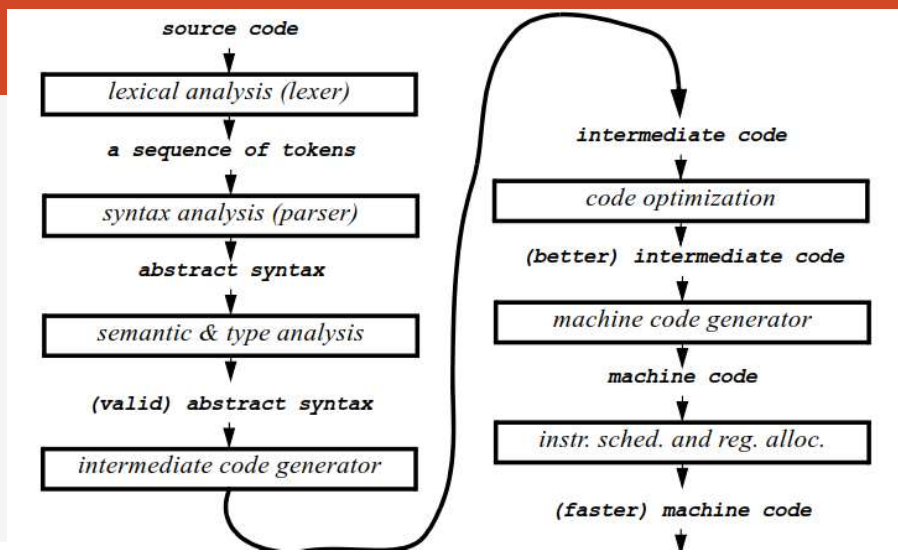
Many CPUs have separate parts for

- Integer Arithmetic
- Floating Point Arithmetic

So these operations can often run in parallel to increase the speed.

Compilers typically generate code into object files which are then linked together by a Linker program.

Compilation Phases



What Compilers Do (and their difference)

Compilers may generate three types of code:

Pure Machine Code

- Machine instruction set without assuming the existence of any operating system or library.
- Mostly being OS or embedded applications.

Augmented Machine Code

- Code with OS routines and runtime support routines.

Virtual Machine Code

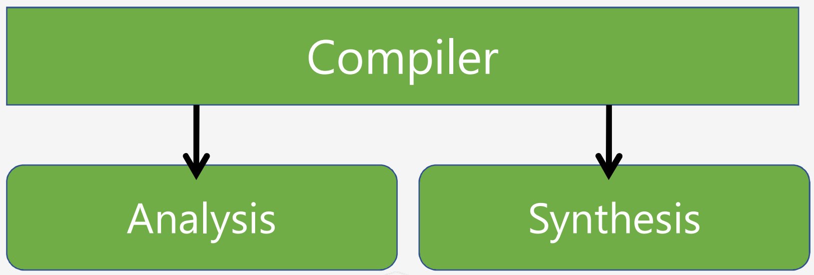
- Virtual instructions, can be run on any architecture with a virtual machine interpreter or a just-in-time compiler
- E.g. Java

Another way that compilers differ from one another is in the format of the target machine code they generate:

- Assembly or other source format
- Relocatable binary
  - Relative address
  - A linkage step is required
- Absolute binary
  - Absolute address
  - Can be executed directly

The Structure of a Compiler (1/7)

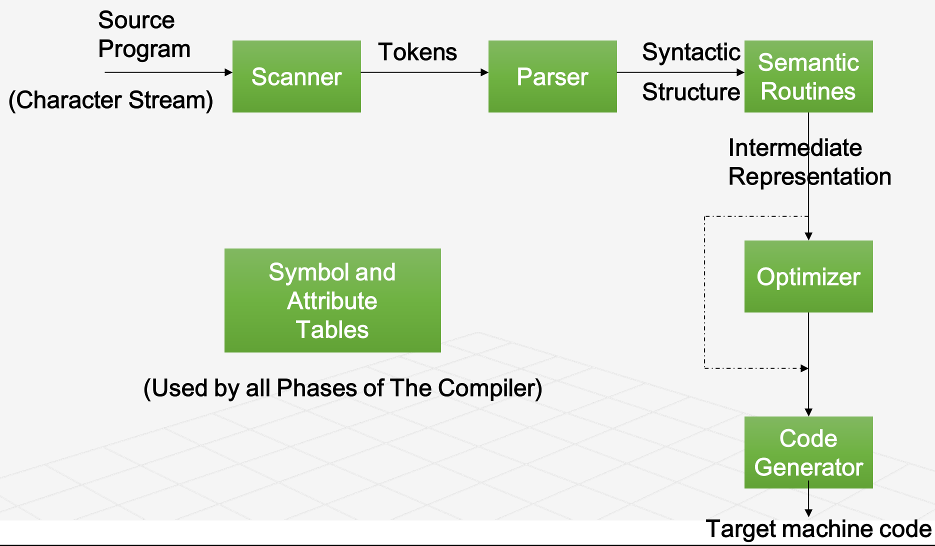
Any compiler must perform two major tasks



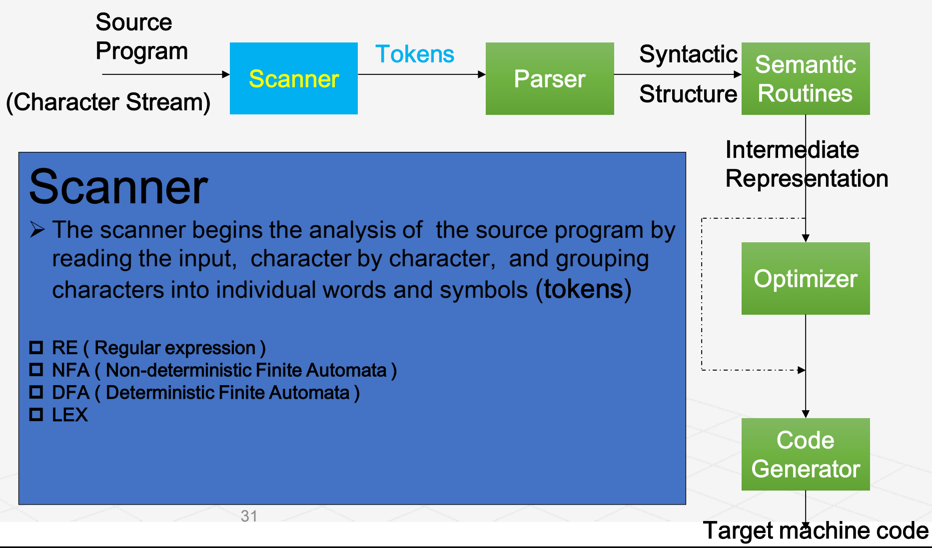
Analysis of the source program

Synthesis of a machine-language program

The Structure of a Compiler (2/7)



The Structure of a Compiler **(3/7)**



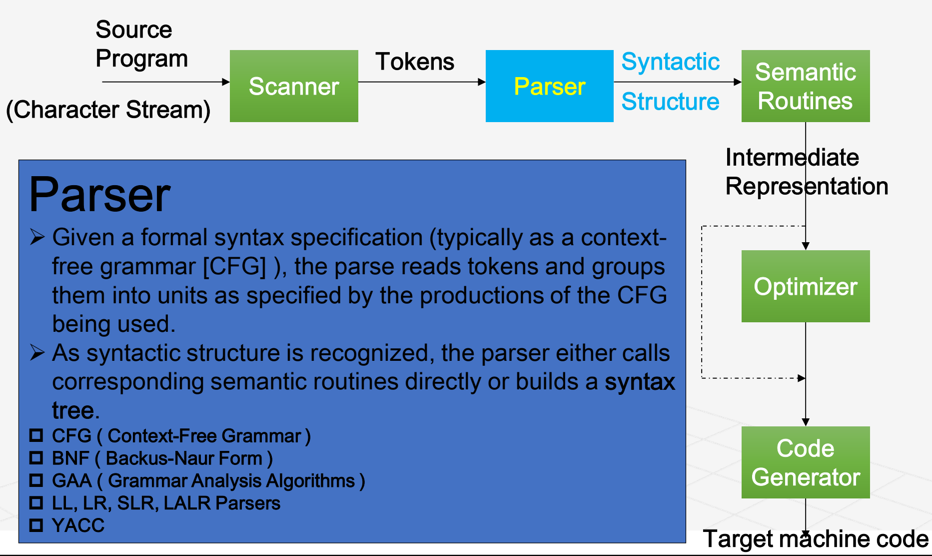
Scanner

- The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (**tokens**)

- RE (Regular expression)
- NFA (Non-deterministic Finite Automata)
- DFA (Deterministic Finite Automata)
- LEX

31

The Structure of a Compiler **(4/7)**

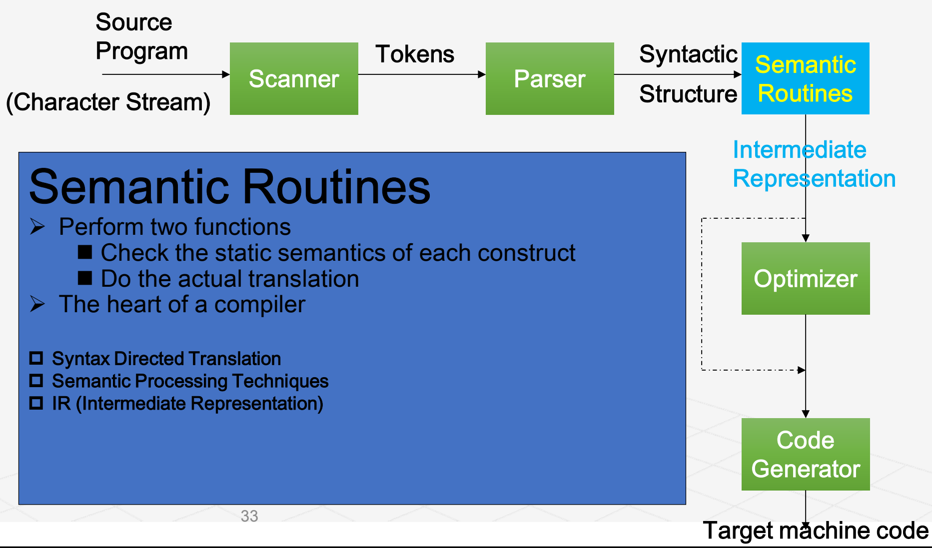


Parser

- Given a formal syntax specification (typically as a context-free grammar [CFG]), the parse reads tokens and groups them into units as specified by the productions of the CFG being used.
- As syntactic structure is recognized, the parser either calls corresponding semantic routines directly or builds a **syntax** tree.

- CFG (Context-Free Grammar)
- BNF (Backus-Naur Form)
- GAA (Grammar Analysis Algorithms)
- LL, LR, SLR, LALR Parsers
- YACC

The Structure of a Compiler (5/7)



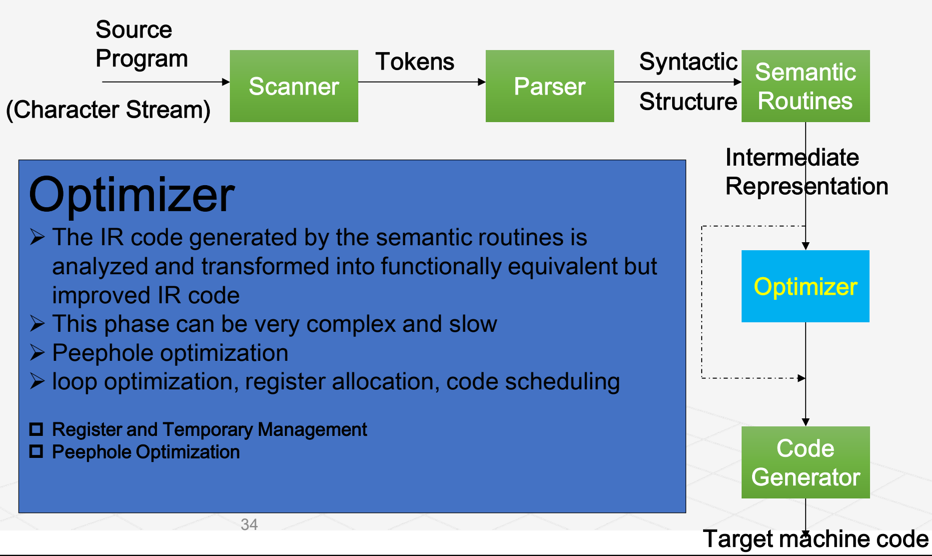
Semantic Routines

- Perform two functions
  - Check the static semantics of each construct
  - Do the actual translation
- The heart of a compiler

- Syntax Directed Translation
- Semantic Processing Techniques
- IR (Intermediate Representation)

33

The Structure of a Compiler (6)



Optimizer

- The IR code generated by the semantic routines is analyzed and transformed into functionally equivalent but improved IR code
- This phase can be very complex and slow
- Peephole optimization
- loop optimization, register allocation, code scheduling

- Register and Temporary Management
- Peephole Optimization

34

Peephole Optimization

In compiler theory, peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions.

Some common techniques (replacement rules) applied in peephole optimization include:

- Null sequences – Delete useless operations.
- Combine operations – Replace several operations with one equivalent.
- Algebraic laws – Use algebraic laws to simplify or reorder instructions.
- Special case instructions – Use instructions designed for special operand cases.
- Address mode operations – Use address modes to simplify code.
- There can be other types of peephole optimizations

Example (Peep hole optimization)

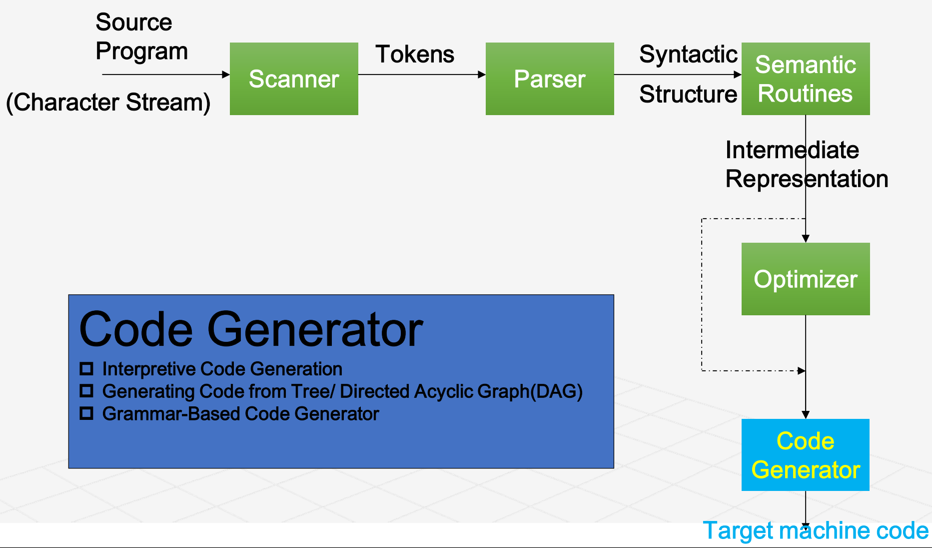
```
...
load 1
load 1
mult
...
```

can be replaced by

```
...
load 1
dup
mult
...
```

The Assumption is that **dup** (duplicate) is faster/optimized than “**aload**”

The Structure of a Compiler **(7/7)**



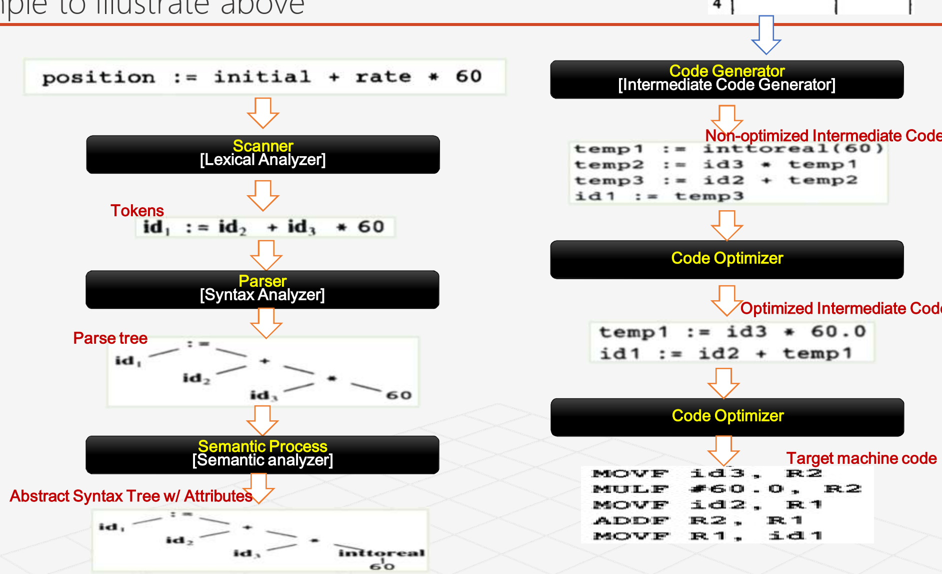
Code Generator

- Interpretive Code Generation
- Generating Code from Tree/ Directed Acyclic Graph(DAG)
- Grammar-Based Code Generator

The Structure of a Compiler (8)
Example to illustrate above

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...
4		



Thank You

➔NEXT

Designing a Simple Compiler