Compiler Theory SCS3211

Dr. Dinuni Fernando

Senior Lecturer

Course Information

Instructor : Dr. Dinuni Fernando
Ms. Yasodha Vimukthika

Office : W313
Email : dkf@ucsc.cmb.ac.lk / rdy@ucsc.cmb.ac.lk

BINGHAMTON UNIVERSITY

STATE UNIVERSITY OF NEW YORK

Dr. Dinuni Fernando

- Holy Family Convent – Colombo 4
- Graduate from UCSC 2014 – CS major, 4 year degree, 1st class honors
  - 2009-2014
- PhD – SUNY Binghamton, NY, USA 2019 GPA (3.9/4)

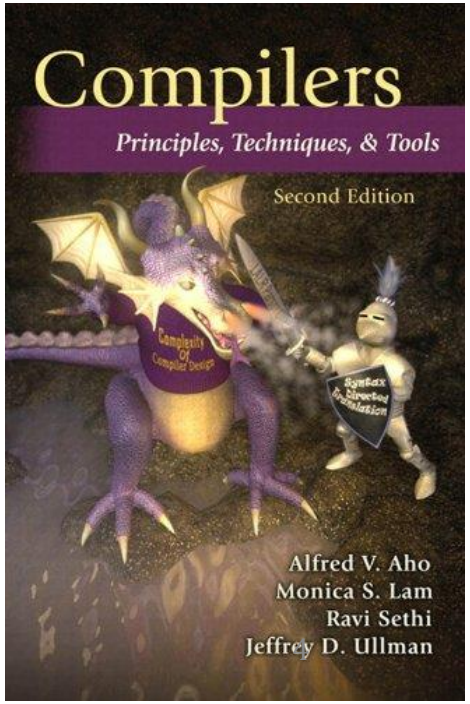
- Research Interests
  - Virtualization – Cloud computing
  - Blockchain and security
  - Software-defined networking

Course Information (cont'd)

Recommended text book

Compilers: Principles, Techniques, and Tools 2nd Edition

by Alfred Aho (Author), Jeffrey Ullman (Author), Ravi Sethi (Author), Monica Lam (Author)



Prerequisites

- Automata Theory course SCS2212.
  - Definition of grammars
  - Finite Automata
  - Parse trees
  - Regular languages
  - Context-free languages.
  - Grammar Transformation
  - Push-down automata
- Proficient with Programming in C.
- Comfortable working and programming in the Unix environment.

Learning Objectives of the course

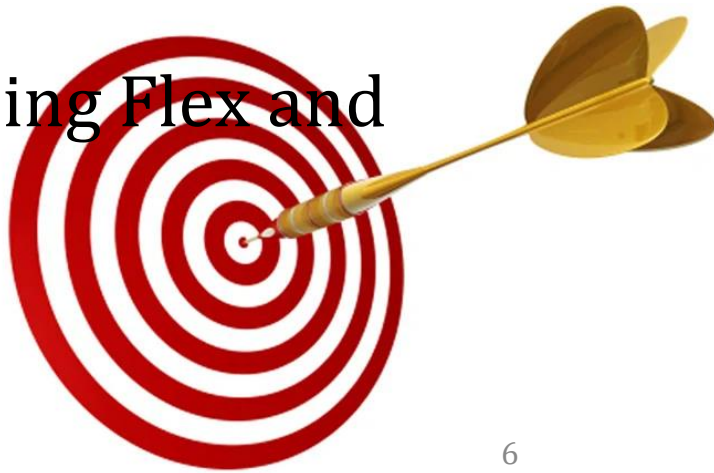
LO1: Understand what is a compiler and its main components.

LO2: Understand and differentiate different language implementations along with their pros and cons.

LO3 : Write a simple application in MIPS assembly language.

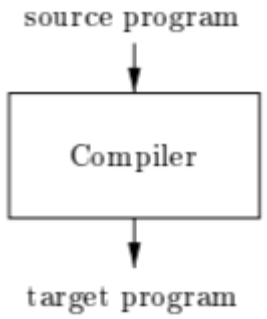
LO4 : Convert simple high level language into MIPS assembly language.

LO5: Implement a compiler for a simple language using Flex and Bison.



What is a Compiler ?

What is a Compiler ?

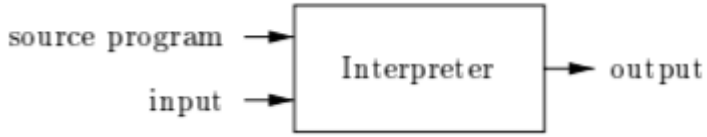


- Software systems that do translation are called compilers.
  - Programming languages are notations that describe computations to people and machines.
  - Before a program to be run, first it must translate into a form in which computers can execute.
- Compiler - a program that can read a program in one language (source) and translate it into equivalent program in another language (target).
  - Reports any errors in source program that detects during the translation process.

What is a Compiler ? (Cont'd)

- If the target program is an executable machine language program, then it can be called by user to process inputs and produce outputs.

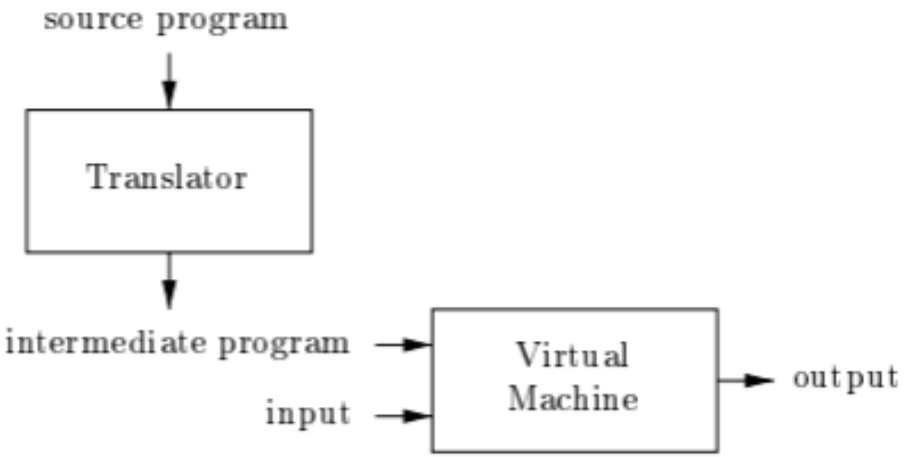
- Interpreter - another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in source program on input given by user.



How about Java language processor ?

How about Java language processor ?

- Combines compilation and interpretation.
- A Java source code first compiled into an intermediate version (bytecode).
- Bytecode are then interpreted by a VM.
- This is an advantage of this setting such that bytecode compiled on one machine can be interpreted on another.

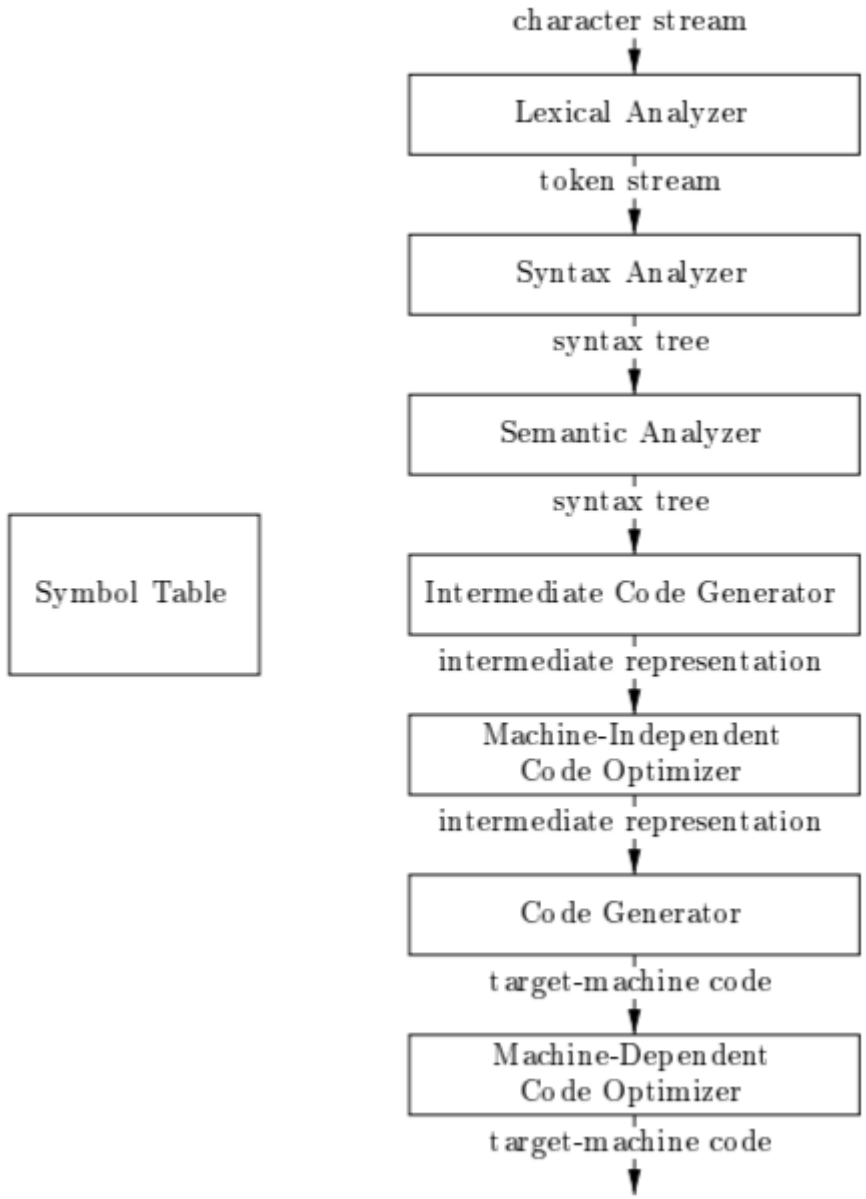


Structure of a Compiler

- Divide into 2 parts - Analysis and Synthesis
  1. Analysis (frontend of the compiler)-
     a. break up the source program into constituent pieces and imposes a grammatical structure to them.
     b. use this structure to create an immediate representation of the source program.
     c. if analysis detects either syntactically ill or semantically unsound, it must provide informative messages to take corrective action.
     d. Collect information about the source program and stores in symbol table.
  2. Synthesis (backend of the compiler) -
     a. construct the desired target program from the intermediate representation and information from the symbol table.

Phases of a Compiler

- Operates as a sequence of phases - each transforms one representation of source program to another.
- Symbol table - stores information about entire source program - used by all phases of the compiler.
- Optimizations are optional and purpose is to produce intermediate representations.



Lexical Analysis/ scanning

- First phase of a compiler.
- Reads the stream of characters making up the source program and group the characters into meaningful sequences (lexemes)
- For each lexeme, lexical analyzer produces as output a token of the form

  `<token-name,attribute-value>`

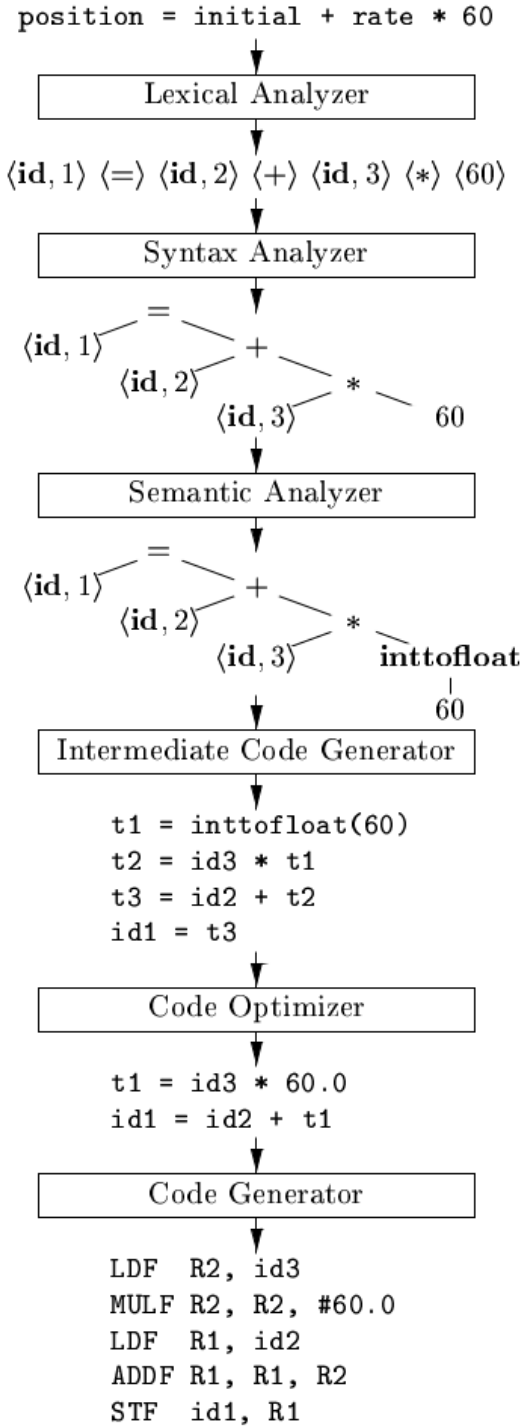
- In the token,
  - `token-name` : abstract symbol used during syntax analysis
  - `attribute-value` : points to an entry in the symbol table for this token (this info required for semantic analysis and code generation).

```
position = initial + rate * 60
```

Lexical Analysis/ scanning (cont'd)

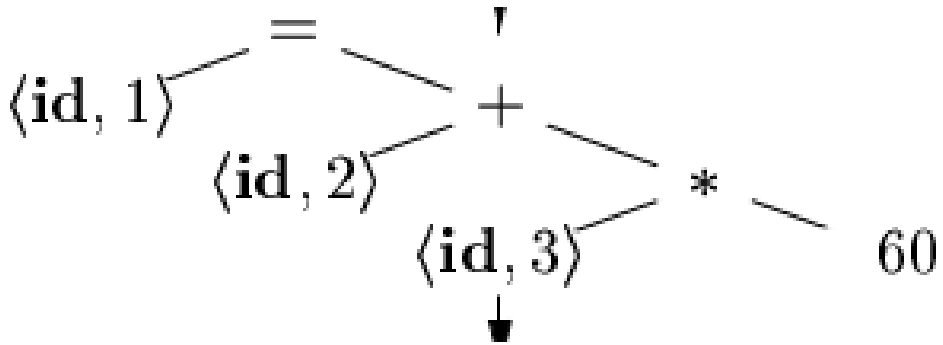
1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Syntax Analysis / Parsing

- Second phase of the compiler.
- Uses first components of tokens produced by lexical analyzer to create a tree-like intermediate representation (grammatical structure of token stream).
- Syntax tree - interior node represents an operation and children nodes represent arguments



Semantic Analysis

- Uses syntax tree and info in the symbol table to check source program for semantic consistency with language definition.
- Also gather type info and save it in syntax tree or symbol table. (for intermediate code generation)
- One of important part is **type checking** - compiler checks each operator has matching operands.
  - eg: For type conversions - if binary arithmetic operator to either float or int. if operator is applied to float and integer, compiler may convert integer into float.
  - if extra node for operator inttofloat includes in semantics analyzer - explicitly converts into float.

Intermediate Code Generation

- low-level or machine-like intermediate representation
  - easy to produce
  - easy to translate into the target machine.
- Three-address code
  - sequence of assembly like instructions with three operands per instruction
  - each operand can act like a register

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

- At most one operator on the right hand side
- Fix the order which operations are to be performed
- Multiplication precedes addition
- Compiler must generate temporary name to hold the value computed by instruction
- Some instructions have fewer than 3 operands eg: 1 and 4

Code Optimization

- Machine - independent
- Attempt to improve intermediate code for a better(fast) target code
  - Shorter code
  - Target code that consumes less power

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

→

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation

- Takes input an intermediate representation of the source program and maps it into the target.
- If target language is machine code - registers or memory locations are selected for each of the variables used by the program.
- Then intermediate instructions are translated into sequences of machine instructions to perform the same task.
- Crucial aspect of code generation is to judicious assignment of registers to hold variables.

Code Generation (Cont'd)

```
t1 = id3 * 60.0
id1 = id2 + t1
```

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

1. Assume R1 and R2 are registers.
2. first operand of each instruction specifies a destination.
3. F in each instruction deals with floating-point numbers.

1. Above code loads contents of address id3 into R2
2. multiplies its with float constant 60.0
3. # - treat as an immediate constant
4. 3rd instruction moves id2 to R1 and 4th adds to it previously computed R2.
5. Finally, value in R1 is stored into address of id1.

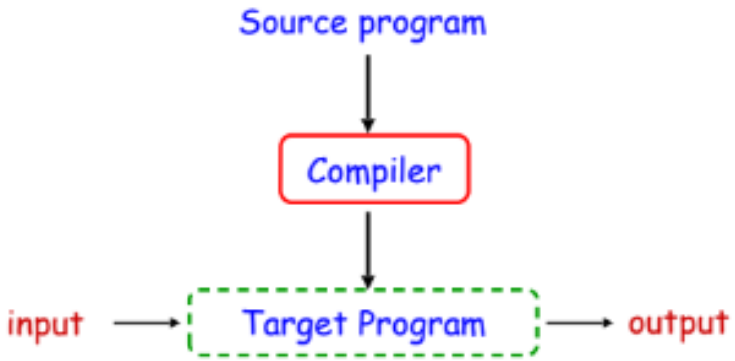
Symbol-Table Management

- Is a data structure containing a record for each variable name with fields for the attributes of the name.
- DS should be designed to allow the compiler to find the record of each name quickly and to store/retrieve data from record quickly.

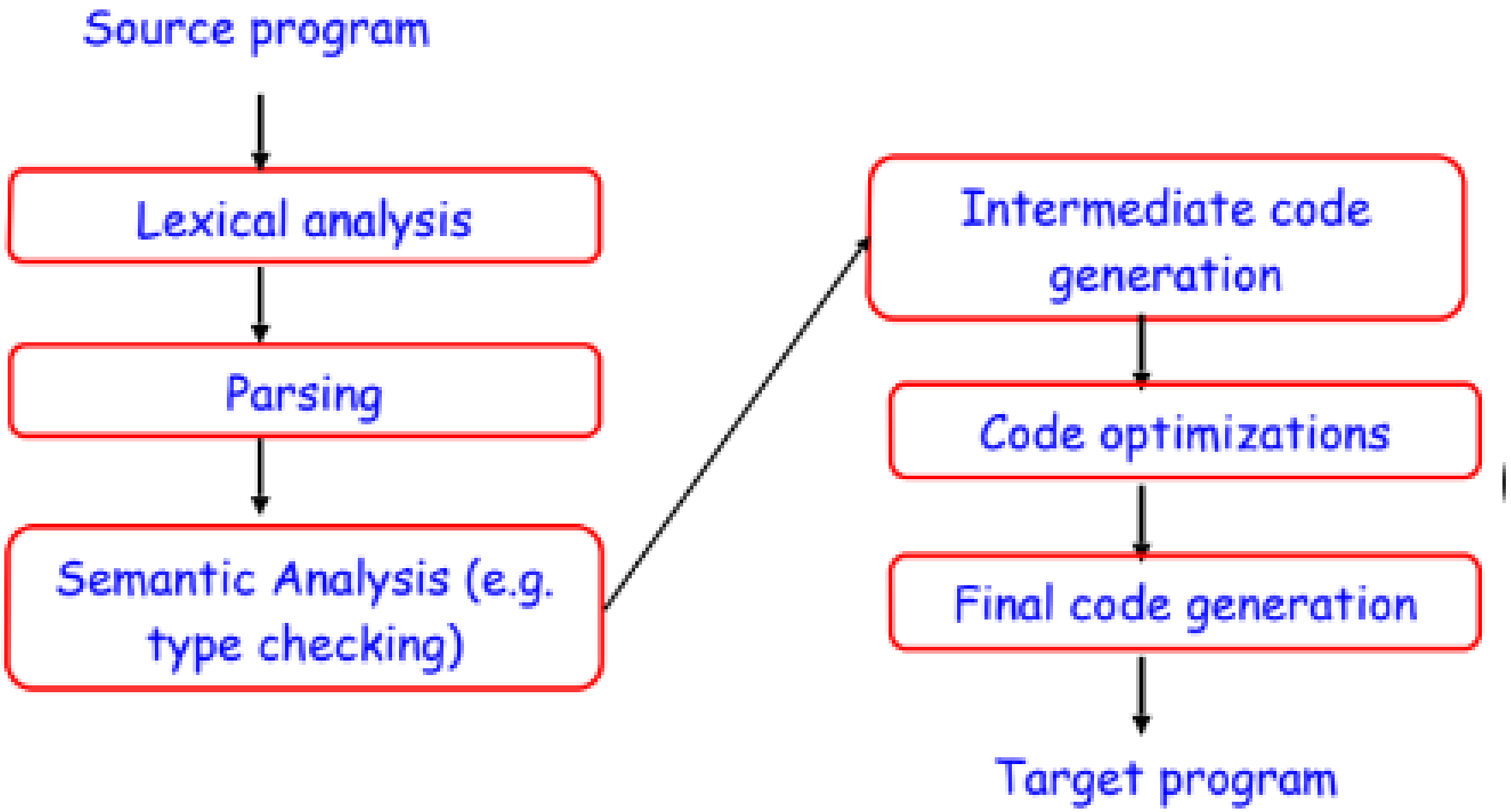
Summary - Compiler

- Translates source code into target code.
- The user may execute the target code.

- The source and the target programs must be semantically equivalent - compilation process must be meaning preserving.



Summary - Phase of Compilation



Thank you!