Introduction

Sudakshina Dutta

IIT Goa

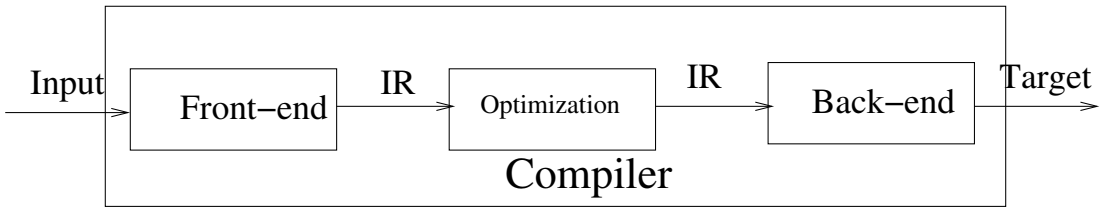
21^{st} January, 2022

Introduction

- A Compiler is a program that can read a program in one language (source) and translate it into an equivalent program in another language (target)

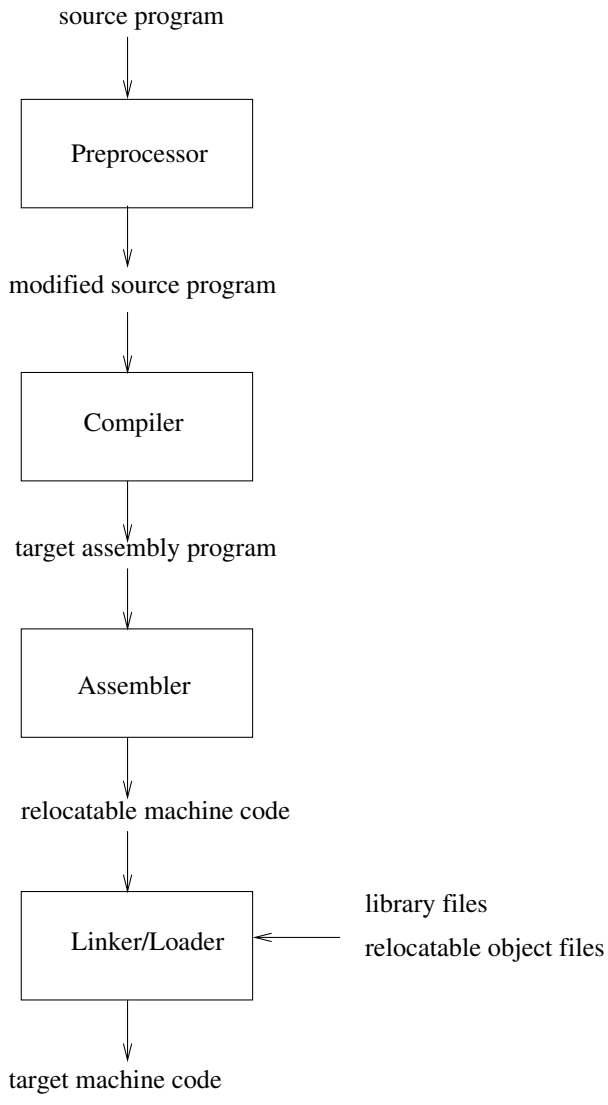
- We use compilers for generating machine language program from the input high-level language program
- Machine language program is called by user to generate output from input

Compilers



- The front-end is also called *analysis part*
- The back-end is also called *synthesis part*

The Language Processing System



The Language Processing System

- The language processing system constitutes several other programs e.g, Preprocessor, Assembler, Linker/Loader
- **Preprocessor :** A program that processes its input program to produce output that is used as input to some subsequent programs like compilers. Tasks include macro substitution, textual inclusion of other files, and conditional compilation or inclusion.

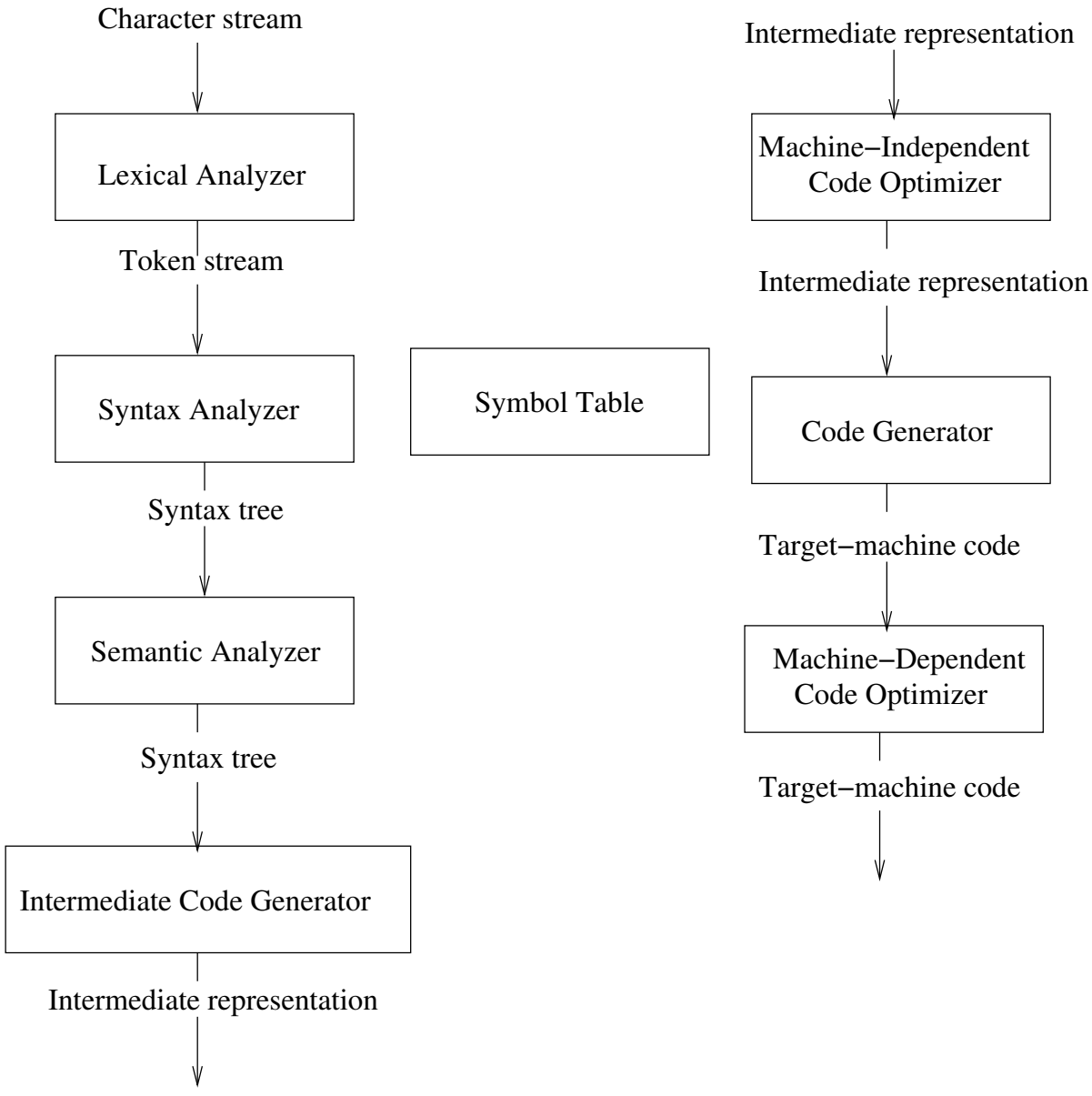
The Language Processing System

- **Assembler :** Compilers may produce assembly language program as output which is processed by a program called *assembler* that produces relocatable machine code.
  - Relocatable code is software whose execution address can be changed

The Language Processing System

- **Linker :** Large programs are often written in pieces. External memory addresses are resolved.
- **Loader :** It puts together all the executable object files into memory for execution
- c program $\rightarrow$ [compiler] $\rightarrow$ objectFile $\rightarrow$ [linker] $\rightarrow$ executable file (say, a.out)
- execute in command line ./a.out $\rightarrow$ [Loader] $\rightarrow$ [execve] $\rightarrow$ program is loaded in memory

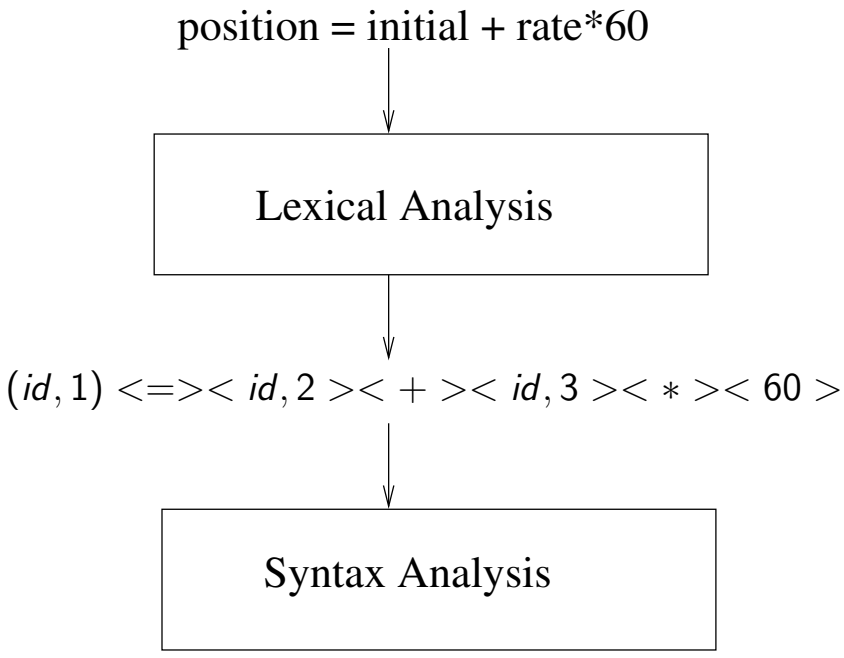
The Phases of a Compiler



The Phases of a Compiler

- Analysis and Synthesis
- Analysis - Breaks up the source program and impose grammatical structures on them (**front-end**)
- Synthesis - Constructs the target program from intermediate representation (**back-end**)
- A data structure called **Symbol table** is used

Lexical Analysis



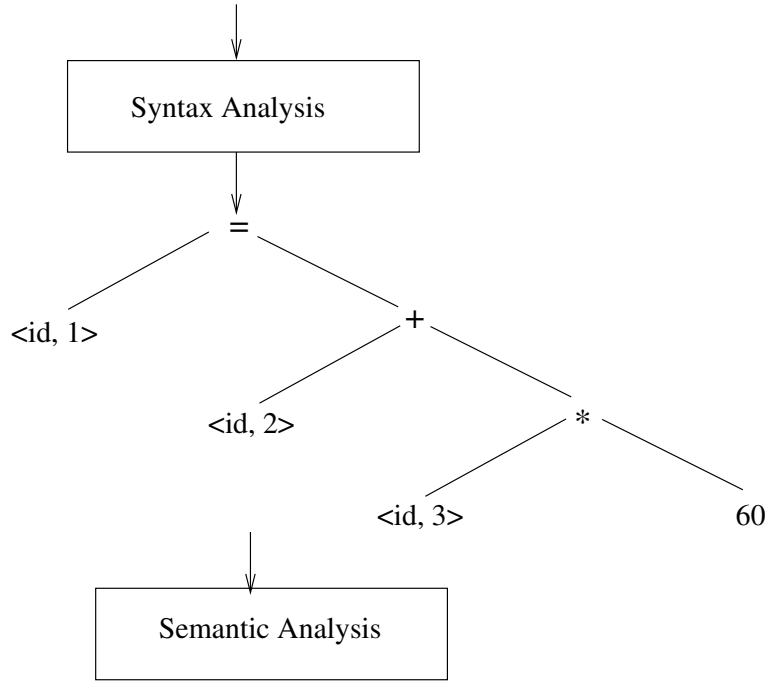
- First phase lexems/tokens
- Groups the characters into meaningful sequences called *lexems*
- For each lexeme, the LA produces the token <token-name, attribute-value>
- A data structure called **Symbol table** is used

Lexical Analysis

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

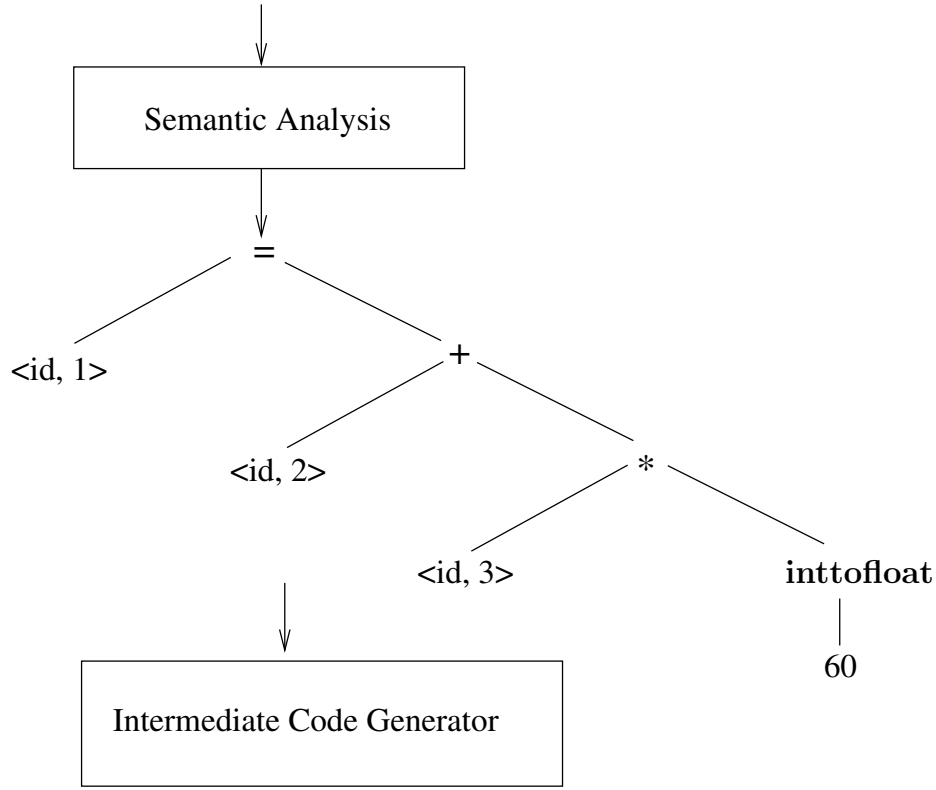
- *position* is mapped to a token <id, 1> where **id** stands for identifier and 1 points to symbol table entry for position
- *, + map into the token <+>, <*>, respectively

Syntax Analysis



- Input to this phase is $\langle id, 1\rangle$, $\langle id, 2\rangle$, $\langle id, 3\rangle$, $\langle +\rangle$, $\langle *\rangle$, $\langle =\rangle$
- After this phase, a tree-like intermediate form is output which represent the grammatical structure of the token stream
- The interior nodes represent operation and the leaf nodes represent arguments of the operation
- The ordering is consistent with precedence

Semantic Analysis



- Uses syntax tree and the symbol table for checking semantic consistency
- Type checking is one of the major part — the analyzer checks whether each operator has matching operands
- Type-casting, coercion are performed in this phase

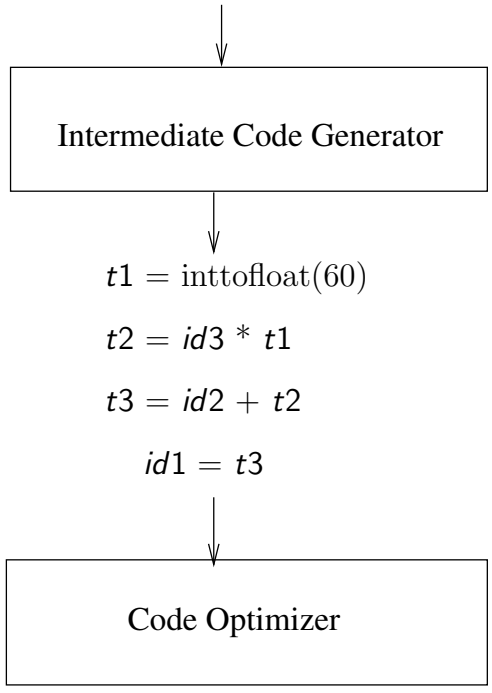
Semantic Analysis

- *position*, *initial*, *rate* are floating point numbers
- Lexeme 60 is an integer — it is coerced to a floating point number
- The information is stored in symbol table or in the syntax tree

Intermediate Code Generation

- Input : Symbol table and intermediate representation
- Output : Three-address code/quadruple/triple/abstract syntax tree etc.
- Two important properties : easy to produce and easy to translate it to target machine code

Intermediate Code Generation

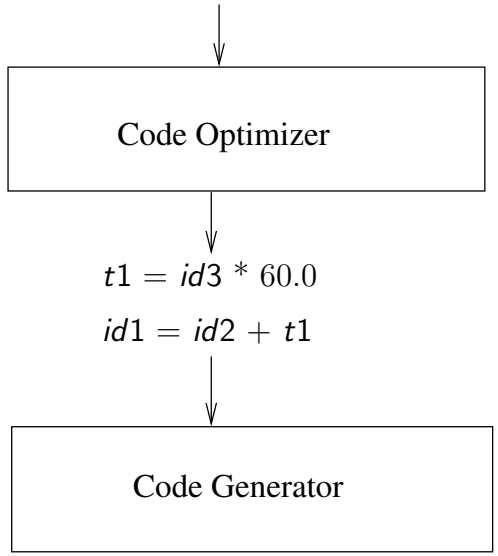


- Example - Three address code
- Three operands per instruction
- At most one operator at the right hand side

Code Optimization

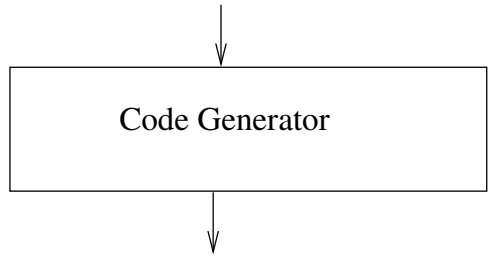
- Machine independent code-optimization phase attempts to improve the intermediate code so that better code is generated in terms of time and space
- A significant amount of time is spent on this phase
- Mostly simple optimizations are tried which improves the code without slowing down compilation

Code Optimization



- Conversion of 60 from integer to float to eliminate *inttofloat* operation
- A shorter sequence is sorted out

Code Generation



- Input : intermediate representation, Output : target code
- There is also a phase called *machine-dependent optimization* where some optimizations specific to the target machine is performed