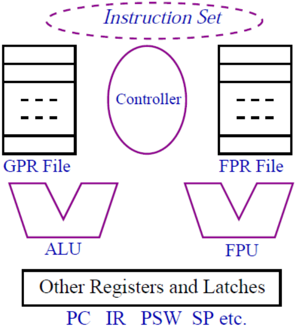
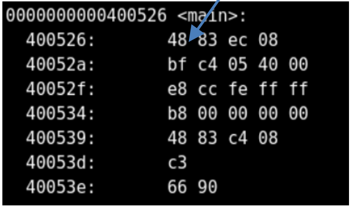
Introduction

- Programs are the instructions written in high-level languages.
    - Source code -- User convenience
- Computer executes the programs written in machine language
    - Machine code --- machine convenience

Computer Architecture



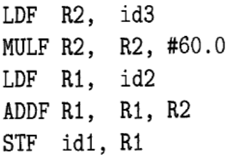
Op code



```
0000000000400526 <main>:
  400526:       48 83 ec 08
  40052a:       bf c4 05 40 00
  40052f:       e8 cc fe ff ff
  400534:       b8 00 00 00 00
  400539:       48 83 c4 08
  40053d:       c3
  40053e:       66 90
```

Machine code

Machine dependent



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

Assembly code

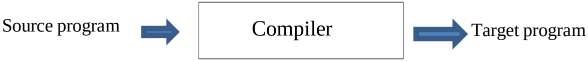
Introduction

- Programs are the instructions written in high-level languages.
    - Source code -- User convenience
- Computer executes the programs written in machine language
    - Machine code --- machine convenience

- Programming in machine language requires memorization of the binary codes — difficult for program-writers
- Hence, the requirement of **Compilers**

Introduction

- A Compiler is a **software**
- Task of a compiler
    - Read a program in one language (**source**) and
    - Translate it into an equivalent program in machine language (**target**)



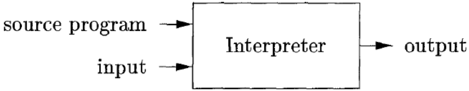
Source program → Compiler → Target program

- Report any errors in the source program that it detects during the translation process.

- We use compilers for generating **target machine** language program from the input high-level language program
- Target program is used by user to generate output from input

Interpreter

- An interpreter is another common kind of language processor.
- Instead of producing a target program as a translation,
  - interpreter directly **translates and executes** the instructions specified in the source program
  - executes the source program statement by statement



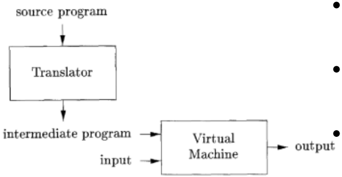
source program

input

Interpreter

output

The machine-language target program produced by a compiler is usually much faster than an interpreter.



source program

Translator

intermediate program

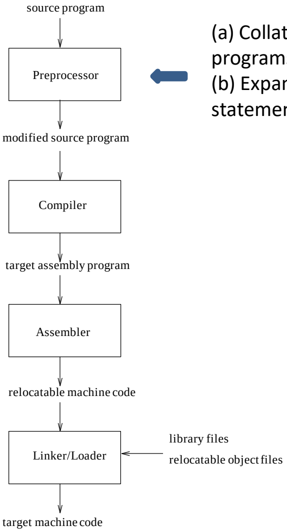
input

Virtual Machine

output

- A Java source program may first be compiled into an intermediate form called bytecodes.
- The bytecodes are then interpreted by a virtual machine.
- Bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

Steps for target code generation



source program

Preprocessor

modified source program

Compiler

target assembly program

Assembler

relocatable machine code

Linker/Loader

library files

relocatable object files

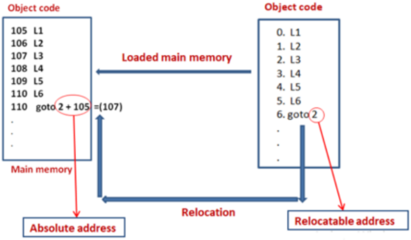
target machine code

(a) Collating and collecting multiple source programs (stdio.h etc) – file inclusion
(b) Expand macros, into source language statements – macro substitution

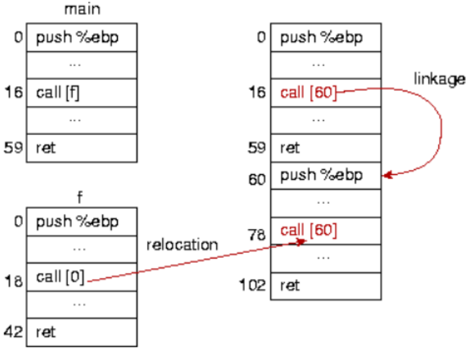
(a) Large programs are often written and compiled in pieces

(b) the relocatable machine code may have to be linked together **with other relocatable object files and library files** into the code that actually runs on the machine.

Linking and relocation



Address relocation



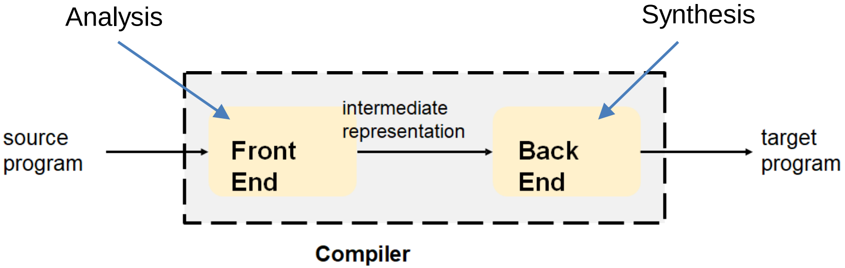
Linking time address relocation

The linker resolves external memory addresses, where the code in one file may refer to a location in another file.

Steps for target code generation

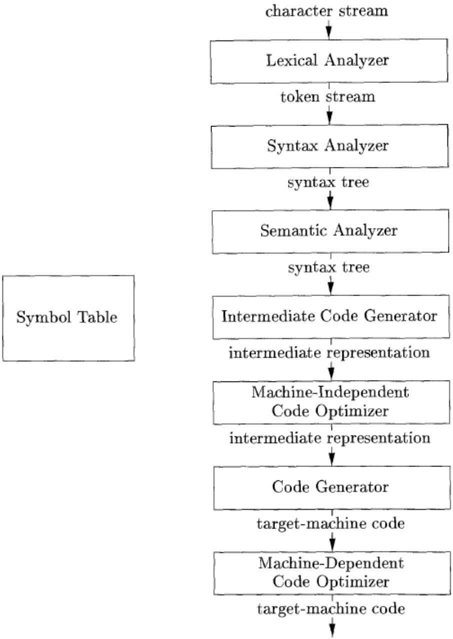
- **Loader :** It puts together all the executable object files into memory for execution
- c program → [compiler] → objectFile → [linker] → executable file (say, a.out)
- execute in command line ./a.out → [Loader] → [execve] → program is loaded in memory

Compiler structure

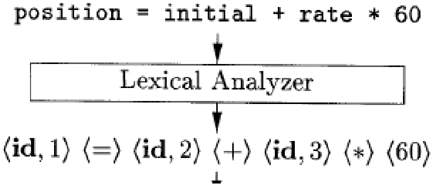


- Analysis and Synthesis
- Analysis - Breaks up the source program and imposes grammatical rules on them (**front-end**)
    - Generates IR
    - Detects errors
    - Constructs **Symbol table**
- Synthesis - Constructs the target program from intermediate representation & the symbol table (**back-end**)

The Phases of a Compiler



Lexical Analysis



`position = initial + rate * 60`

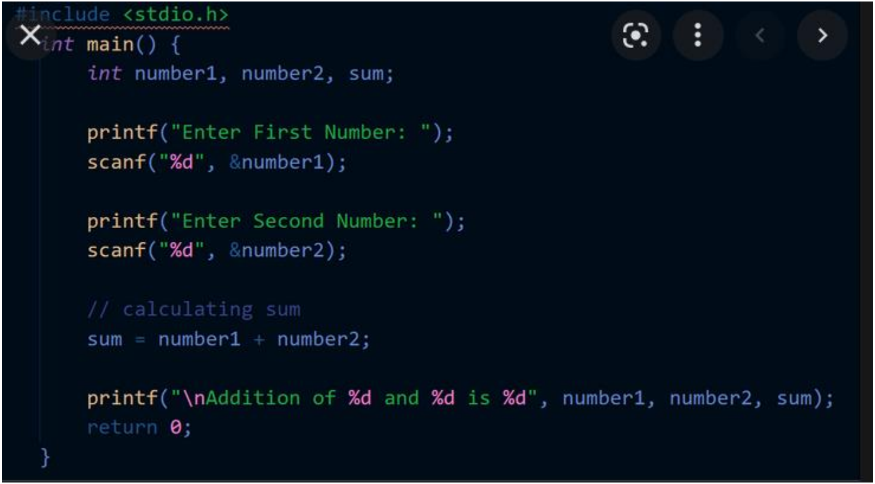
Lexical Analyzer

$\langle \mathbf{id}, 1\rangle \; \langle = \rangle \; \langle \mathbf{id}, 2\rangle \; \langle + \rangle \; \langle \mathbf{id}, 3\rangle \; \langle * \rangle \; \langle 60\rangle$

- Reads the stream of characters making up the source program and groups the characters into meaningful sequences called **lexemes**
- For each lexeme, the LA produces the **token**, (a) passed to the syntax analyzer, (b) inserted in the **symbol table**

  <token-name, attribute-value>

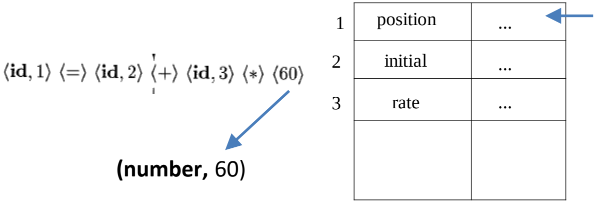
- **token-name** is an abstract symbol that is used during syntax analysis, and the second component **attribute-value** points to an entry in the symbol table for this token
- Blanks separating the lexemes would be discarded by the lexical analyzer.



```c
#include <stdio.h>
int main() {
    int number1, number2, sum;

    printf("Enter First Number: ");
    scanf("%d", &number1);

    printf("Enter Second Number: ");
    scanf("%d", &number2);

    // calculating sum
    sum = number1 + number2;

    printf("\nAddition of %d and %d is %d", number1, number2, sum);
    return 0;
}
```

Lexical Analysis

id is an abstract symbol standing for identifier and 1 points to the symbol table entry for **position**.



$\langle \mathbf{id}, 1\rangle \; \langle = \rangle \; \langle \mathbf{id}, 2\rangle \; \langle + \rangle \; \langle \mathbf{id}, 3\rangle \; \langle * \rangle \; \langle 60 \rangle$

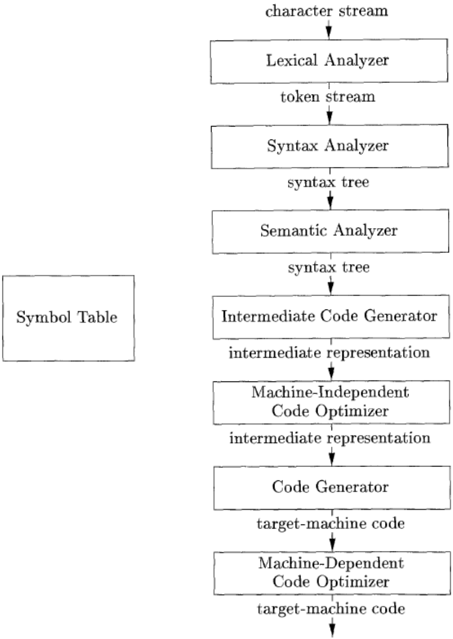
(number, 60)

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

information about the **id**, such as its name and type

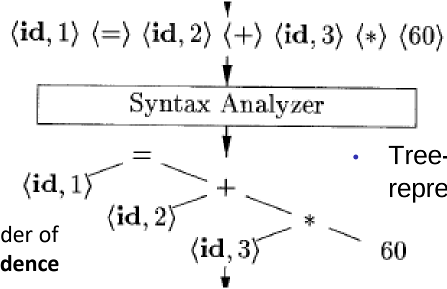
- ▸ ***position*** is mapped to a token <id, 1> where **id** stands for identifier and 1 points to symbol table entry for position
- ▸ *****, **+** map into the token <+>, <*>, respectively

The Phases of a Compiler



Syntax Analysis – Parsing

Context-free grammars are used to represent **grammatical structure** (say, precedence of operations)



$\langle \mathbf{id}, 1\rangle \; \langle=\rangle \; \langle \mathbf{id}, 2\rangle \; \langle+\rangle \; \langle \mathbf{id}, 3\rangle \; \langle*\rangle \; \langle 60\rangle$
Syntax Analyzer
=
$\langle \mathbf{id}, 1\rangle$
+
$\langle \mathbf{id}, 2\rangle$
*
$\langle \mathbf{id}, 3\rangle$
60

- Tree-like intermediate representation
    - Syntax tree

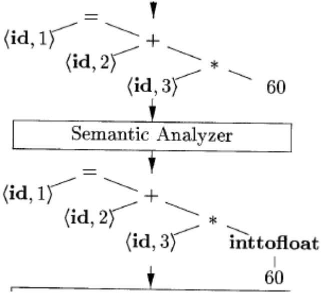
Tree depicts the order of operations – **precedence**

- The **internal nodes** represent **operation** and the **leaf nodes** represent **arguments** of the operation

- The parser uses the **token names** produced by the lexical analyzer to create a tree-like intermediate representation
    - Depicts the **grammatical structure of the token stream**.

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



Binary arithmetic operator may be applied to (i) either a pair of integers or (ii) to a pair of floating-point numbers.
If the operator is applied to a floating-point number and an integer, the compiler may convert the integer into a floating-point number.

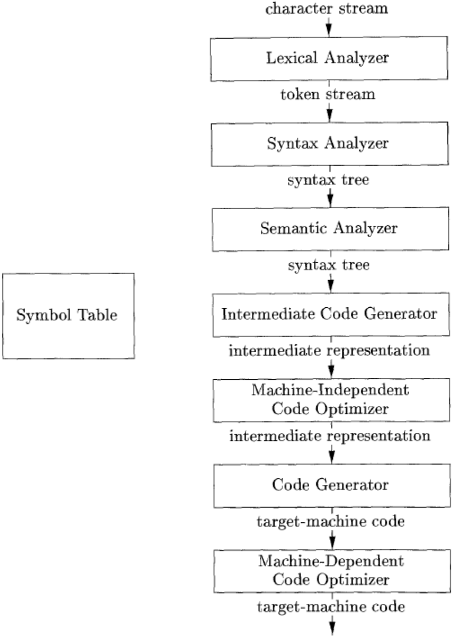
position, *initial*, *rate* are floating point numbers

Lexeme **60** is an integer — it is **type casted** to a floating point number

Type-casting are performed in this phase

The information is stored into syntax tree or in symbol table

The Phases of a Compiler



character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

syntax tree

Intermediate Code Generator

intermediate representation

Machine-Independent Code Optimizer

intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

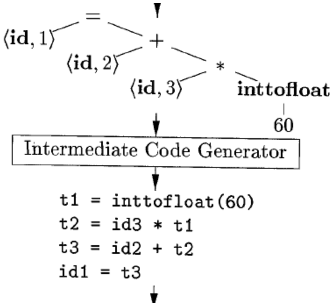
target-machine code

Symbol Table

Intermediate Code Generation

- In the process of translating a source program into target code,
    - compiler constructs multiple **intermediate representations**
        - various forms of Intermediate code (syntax tree etc)
    - Explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine
- (a) IR should be easy to produce and (b) it should be easy to translate into the target machine.
- Three address code (TAC)
    - **Three operands** per instruction
    - **At most one operator** at the right hand side

Intermediate Code Generation



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

The syntax tree fixes the order in which operations are to be done; **the multiplication precedes the addition**.

Notable points:

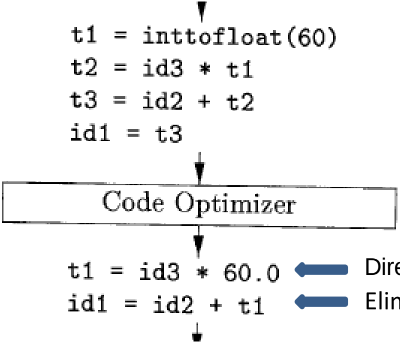
(a) Each three-address instruction has **at most one operator** on the right side.

(b) The compiler must generate a temporary name to hold the value computed by a three-address instruction.

(c) some "three-address instructions" like the first and last in the sequence, above, have **fewer than three operands**

Code Optimization

- Code-optimization phase attempts to improve the intermediate code so that **better code can be generated**
  - Faster
  - Shorter
  - Power optimization



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

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

Directly replace integer 60 by float 60.0

Eliminate t3

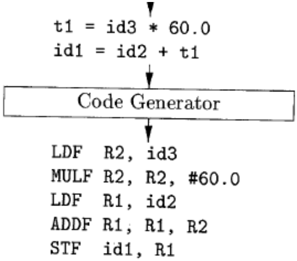
Code Optimization

- A significant amount of time is spent on optimization phase
    - Optimization varies widely
- Mostly simple optimizations aim to improve the target code without slowing down the compilation
- “Optimizing compilers” spend a significant amount of time on this phase

Code Generation

- Input : Intermediate representation, Output : target code **(instruction set)**
- **Registers and memory locations** are selected for **each variable** used by the program
- Then, the **intermediate instructions** are translated into sequences of **machine instructions** that perform the same task.
- Example : above generated code uses only registers R1and R2
    - First operand is the destination

A crucial aspect of code generation is the **judicious assignment of registers** to hold variables



```
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
```