

Compiler

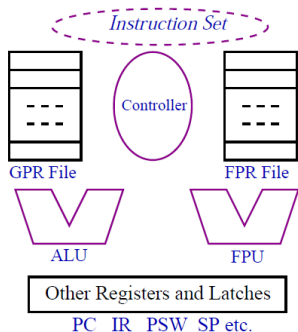
(CS3104)

Introduction 2

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



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



Assembly code

Mnemonics

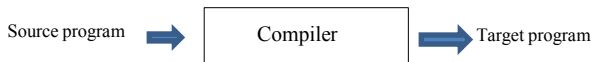
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

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**

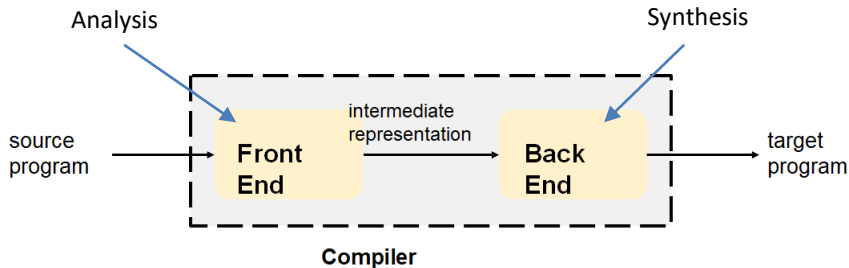
Introduction

- ▶ 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**)



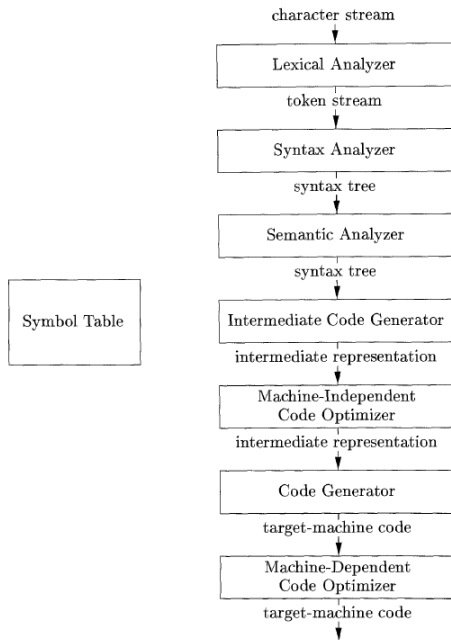
- ▶ Report any errors in the source program that it detects during the translation process.
- ▶ We use compilers for generating **target machine** language program from the input high-level language program
- ▶ Target program is used by user to generate output from input

Compiler structure

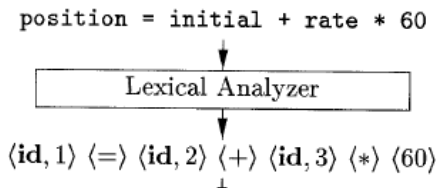


- ▶ Analysis and Synthesis
- ▶ Analysis - Breaks up the source program and imposes grammatical rules on them (**front-end**)
 - ▶ Generates IR
 - ▶ Detects errors
 - ▶ Constructs **Symbol table**
- ▶ Synthesis - Constructs the target program from intermediate representation & the symbol table (**back-end**)

The Phases of a Compiler



Lexical Analysis



- ▶ 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.

Lexical Analysis

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

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

$(\text{number}, 4)$

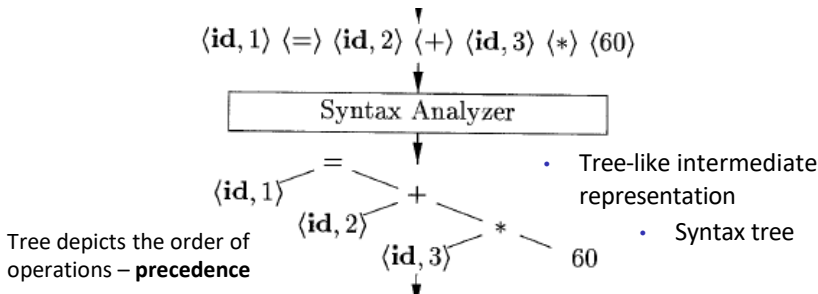
1	position	...
2	initial	...
3	rate	...

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

- ▶ **position** is mapped to a token $\langle \text{id}, 1 \rangle$ where **id** stands for identifier and 1 points to symbol table entry for position
- ▶ *****, **+** map into the token $\langle + \rangle$, $\langle * \rangle$, respectively

Syntax Analysis – Parsing

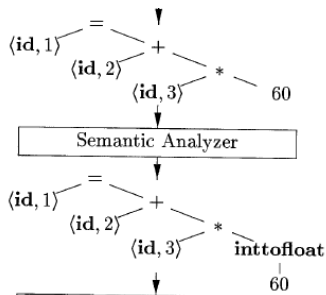
`position = initial + rate * 60`



- The parser uses the **tokens** produced by the lexical analyzer to create a tree-like intermediate representation
 - Depicts the **grammatical structure of the token stream**.
- The **internal nodes** represent **operation** and the **leaf nodes** represent **arguments** of the operation
- **Context-free grammars** are used to represent grammatical structure (say, precedence of operations)

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.

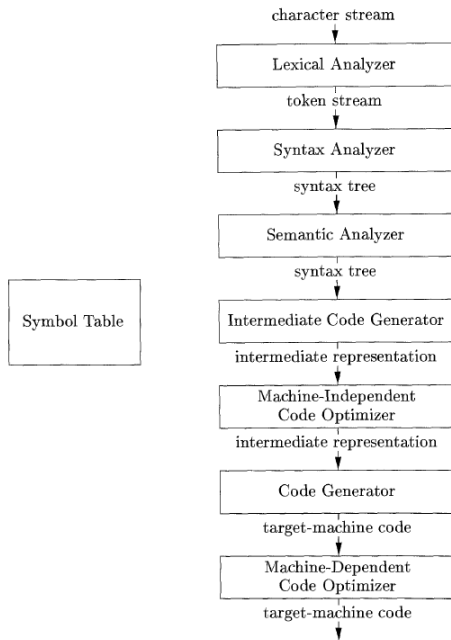
position, initial, rate are floating point numbers

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

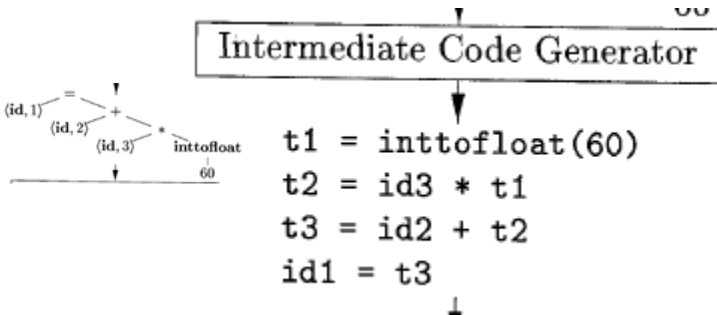
The Phases of a Compiler



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
 - ▶ **Three operands** per instruction
 - ▶ **At most one operator** at the right hand side

Intermediate Code Generation



Notable points:

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

Thus, these instructions fix the order in which operations are to be done; **the multiplication precedes the addition.**

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

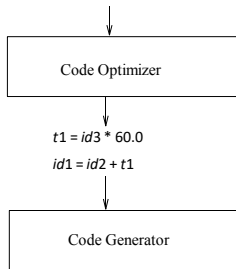
(c) some "three-address instructions" like the first and last in the sequence, above, have **fewer than three operands**

Code Optimization

- ▶ 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

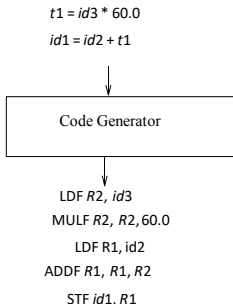
Code Optimization

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



- ▶ 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
- ▶ Registers and memory locations are selected for each variable used by the program
- ▶ Example : above generated code uses only registers *R1* and *R2*
- ▶ First operand is the destination