



Compiler Construction Introduction

Dr. Nadeem Akhtar

Assistant Professor

Department of Computer Science & IT

The Islamia University of Bahawalpur

PhD – Computer Science

IRISA – University of South Brittany – Bretagne - FRANCE

Course Textbooks

- **Compilers – Principles, Techniques and Tools (Second Edition)**
 - Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- **Introduction to Compiler Design** (Springer)
 - Author: Torben Egidius Mogensen

Course Objectives

- **Introduce and Describe the Fundamentals of Compilation Techniques**
- **Assist you in Writing your Own Compiler (or a part of it)**
 - Overview & Basic Concepts
 - Lexical Analysis
 - Syntax Analysis / Syntax-Directed Translation
 - Semantic Analysis
 - Intermediate Code Generation.

What is a Compiler? (1/3)

- **Programming languages:** Notations for describing computations to people and to machines
 - E.g. Java, C#, C++, Perl, Prolog etc.
- A program written in any language must be translated into a form that is understood by the computer
 - This form is typically known as the **Machine Language (ML), Machine Code, or Object Code**
 - Consists of streams of 0's and 1's
- **The program that carries out the translation activity is called the compiler.**

What is a Compiler? (2/3)

A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.

What is a Compiler? (3/3)

Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a **high-level language** tend to be shorter than equivalent programs written in **machine language**.
- The same **high-level language** program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

Basic Concept (1/2)

- The language to be translated: Source language
 - Input code is called the **Source code**
- The language produced: Target language
 - Output code is called **Target code**
- **A compiler is a program that translates source code written in one language into the target code of another language.**

Basic Concept(2/2)

Java

```
sum = 0;  
for (x = 3; x < 5; x++)  
{ cout << "x is " << x;  
  cout << endl;  
  sum += x;  
  a *= b / 2;
```



Machine Language

1	0	0	1	1	1	0	1
0	1	0	1	1	0	1	1
1	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0
1	0	1	0	1	1	0	0
0	0	0	1	1	0	1	1

Why Learn About Compilers?

Why do most computer science institutions offer compiler courses and often make these mandatory?

Some typical reasons are:

- It is considered a topic that you should know in order to be “well-cultured” in computer science.
- A good craftsman should know his tools, and compilers are important tools for programmers and computer scientists.
- The techniques used for constructing a compiler are useful for other purposes as well.
- There is a good chance that a programmer or computer scientist will need to write a compiler or interpreter for a domain-specific language.

Why Compile?

❑ **Executables:** Files that actually do something by carrying out a set of instructions (as compared to files that only contain data)

- E.g., .exe files in Windows
- If you look at their data, it won't make sense

Hex Dump of a
boot loader executable

```
00000000 fc 31 c0 8e c0 8e d8 8e d9 bc 00 7c 85 e5 b1 00 |.....|
00000010 06 b9 00 01 f3 a5 89 fa b1 08 f3 ab fe 45 f2 e9 |.....E..|
00000020 00 8a f6 46 b6 20 75 08 84 d2 78 07 80 4a b6 40 |...F.w...N..|
00000030 8a 56 ba 88 56 00 e8 fc 00 52 b6 c2 07 31 d2 88 |...V.V...R...|
00000040 df 1c 0f a3 56 b6 73 19 8a 07 bf 07 07 b1 03 f2 |e...V.s...|
00000050 ae 74 0a b1 06 f2 ae 83 c7 09 8a 0d 03 cf e8 c5 |t.....|
00000060 00 42 80 c3 10 73 e8 58 2c 7f 3a 06 75 04 72 05 |e...s.X...u..|
00000070 48 74 0f 30 c0 04 b0 88 46 b8 bf b2 07 e8 a6 00 |Ht.O...F...|
00000080 be 7b 07 e8 b2 00 8a 56 b9 4e e8 8a 00 e8 05 b0 |t...V.N...|
00000090 07 e8 b0 00 30 e4 cd 1a 89 07 03 7e bc b4 01 cd |t...G...|
000000a0 16 75 0d 30 e4 cd 1a 39 fa 72 f2 8a 46 b9 e5 16 |w.O...9.r..F..|
000000b0 30 e4 cd 16 88 e0 3c 1c 74 f1 2c 3b 3c 04 76 06 |G...<...s.v..|
000000c0 2c e7 3c 04 77 c9 98 0f a3 46 0c 73 e2 88 46 b9 |c.w...F.s..P..|
000000d0 b4 00 08 8a 14 80 13 3c 04 9c 74 0a c0 e8 04 05 |.....s.t...|
000000e0 b4 07 93 c5 07 80 53 f6 46 b6 40 75 08 b6 00 06 |.....S.F.g...|
000000f0 b4 03 e8 59 00 5a 9d 75 06 8a 56 b8 80 ea 30 b9 |t...Y.r...V..O..|
00000100 00 7c b4 02 e8 47 00 72 96 81 bf fe 01 55 aa 0f |t...G.P...U..|
00000110 85 7c ff ba 85 07 e8 19 00 ff a3 b0 46 e8 24 00 |t...F.s...|
00000120 b0 31 00 d0 e8 17 0f ab 56 0c ba 78 07 e8 eb ff |t...V...|
00000130 85 fe e8 03 00 ba 85 07 ac a0 80 75 05 e8 04 00 |t.....|
00000140 e8 16 24 7f 53 b6 07 00 b4 0e cd 10 5b c3 8a 74 |t.g.S...[...t]|
00000150 01 86 4c 02 b0 01 58 89 e7 16 46 b6 80 74 13 66 |t...L...V...F..t..|
00000160 6a 00 66 ff 74 08 06 53 6a 01 6a 10 89 e8 48 80 |t.g.f.t..S.j...H..|
00000170 cc 40 cd 13 89 fc 5e c3 20 20 e0 0a 44 95 66 61 |e...r...Defal|
00000180 75 6c 74 3a e0 0d 8a 00 05 0f 01 06 07 0b 0c 0e |ult:.....|
00000190 83 a5 a5 a9 0d 0c 0b 0a 09 08 0a 0e 11 10 01 3f |.....f...|
000001a0 bf 44 4f d3 4c 69 6e 75 f8 46 72 65 65 42 53 c4 |t.OO.Lim.FreelB..|
000001b0 66 b6 44 72 69 76 65 20 00 00 80 bf b6 00 00 00 |t.f.Drive.....|
000001c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000001f0 00 00 00 00 00 00 00 00 00 00 00 55 aa |.....U..|
00000200
```



The primary reason for
compiling source code
is to create an
executable ML program

Why Compile?

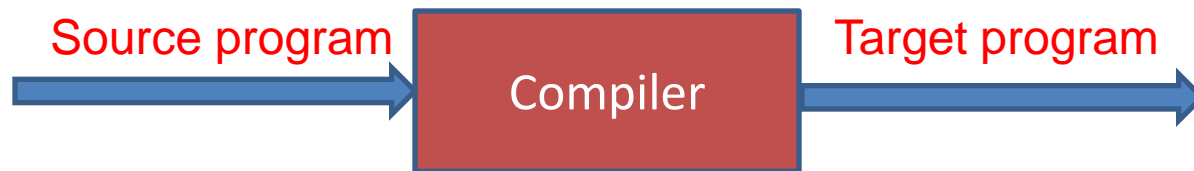
- Once the executable is there, it can be called by the user to process the given inputs to a program and produce the desired outputs



- Initially, there is a **compile phase** which is followed by the **run/execute phase**
 - This approach is used in many languages, e.g., Java, C++, C# etc.
- Interpretation is concept parallel to compilation
 - There is only one **run/execute** phase.

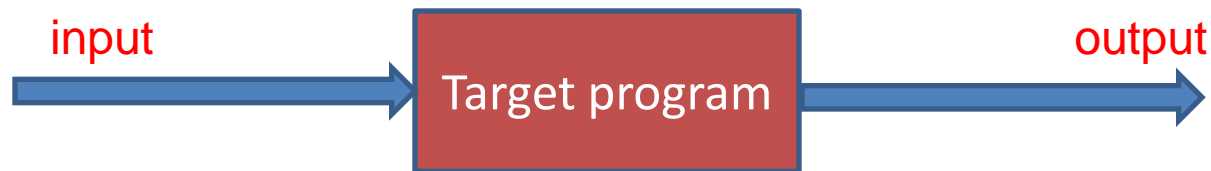
A Compiler

- A compiler is a program that can read a program in one language – and translate it into equivalent program in another language – the target language.



Running the target program

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



Interpreter

- An *interpreter* is 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 the source program on inputs supplied by the user



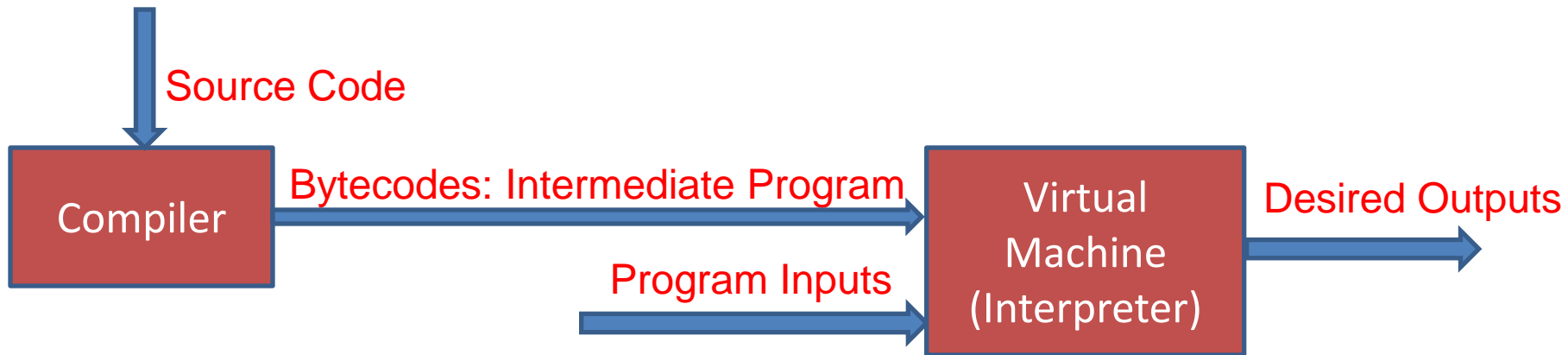
Compiler vs. Interpreter

- **Interpreter:** A program that *doesn't* produce the target executable. It can do three things:
 1. In a line-by-line fashion, **it directly executes the source code** by using the given inputs, and producing the desired outputs
 2. May **translate source code into some intermediate language and then execute that immediately**, e.g., Perl, Python, and Matlab
 3. May also **execute previously stored pre-compiled code**, made by a compiler that is part of the interpreter system, e.g., Java and Pascal
- Some systems, e.g., SmallTalk, may combine 2 and 3.

Interpreter



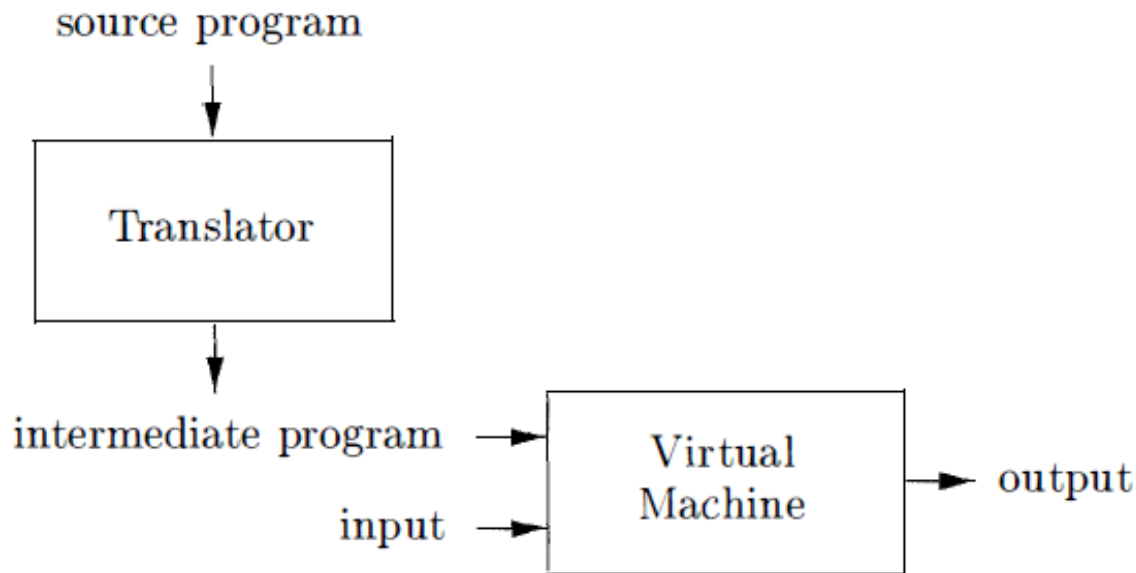
Java combines compilation and interpretation



Portability: Bytecodes compiled on one machine can be interpreted on another one

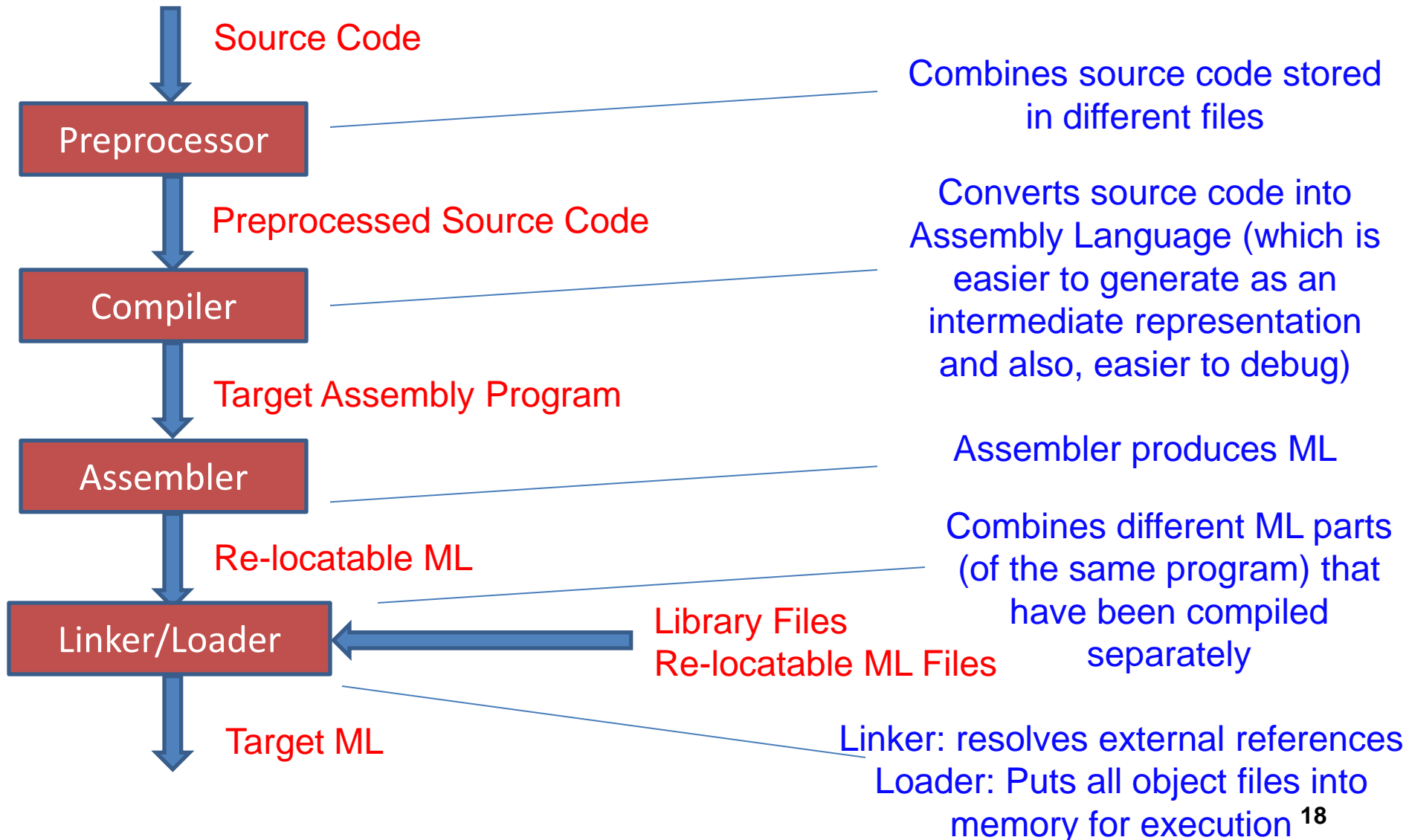
Faster Execution through Just-in-Time (JIC) compilers: translate parts of bytecode into ML immediately before execution

Interpreter



Java language processors combine compilation and interpretation, as shown in above figure. A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine , perhaps across a network.

Language Processing System (1/4)



Language Processing System (2/4)

- **Preprocessor:** A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a **preprocessor**. The preprocessor may also expand short-hands, called **macros**, into source language statements.
- **Compiler:** The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.

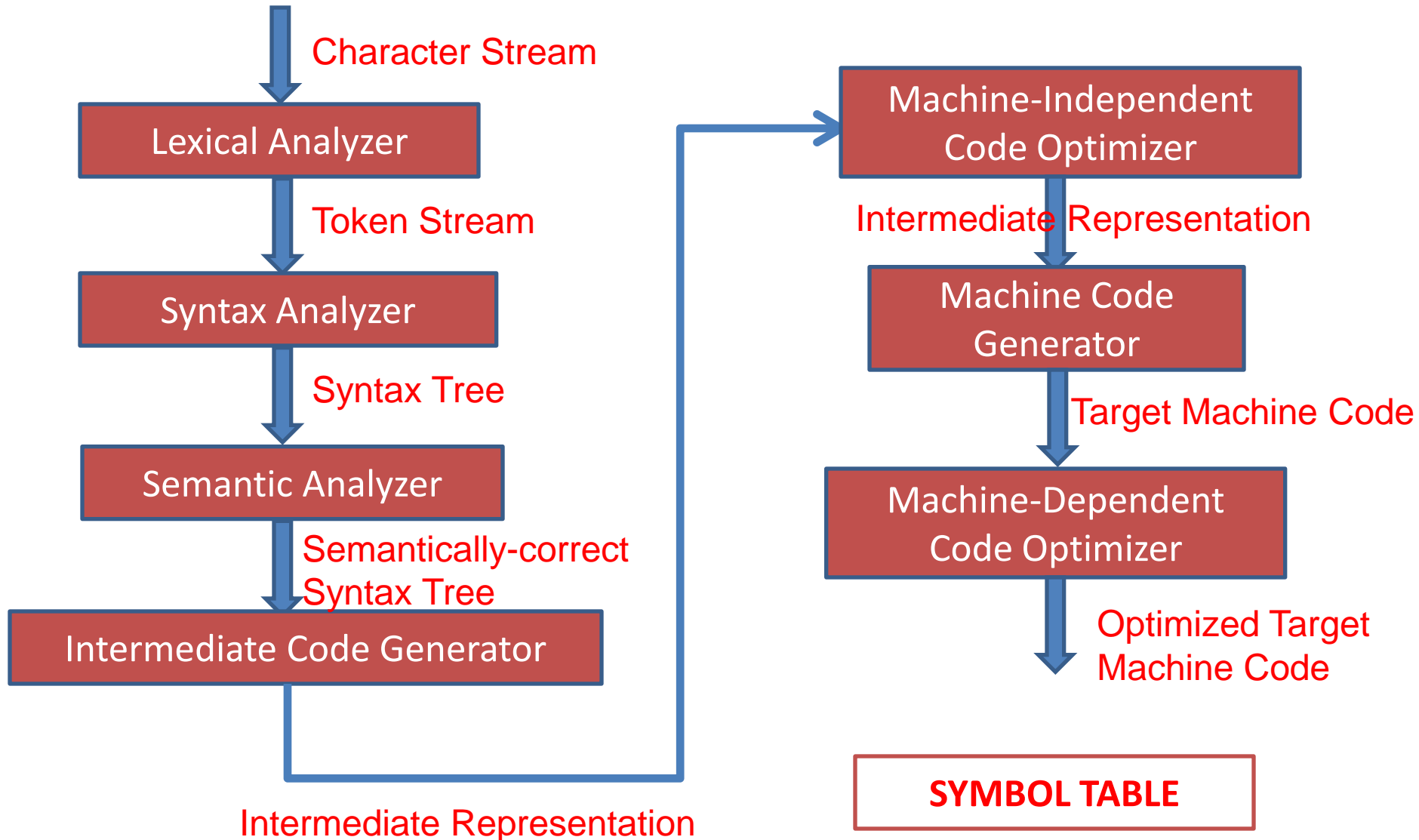
Language Processing System (3/4)

- **Assembler:** The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.
- **Linker:** Large programs are often compiled in pieces, so 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.** The linker resolves external memory addresses, where the code in one file may refer to a location in another file.

Language Processing System (4/4)

- **Loader:** The loader then puts together all of the executable object files into memory for execution.
- **Language Processors.** An integrated software development environment includes many different kinds of language processors such as
 - compilers, interpreters, assemblers, linkers
 - loaders, debuggers, profilers.

Structure of a Compiler (1/4)



Structure of a Compiler (2/4)

Lexical analysis

This is the initial part of reading and analyzing the program text. The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

Syntax analysis (Parsing)

This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing.

Semantic Analysis (Type checking)

This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that does not make sense given the type of the variable, such as trying to use a boolean value as a function pointer.

Structure of a Compiler (3/4)

Intermediate Code Generator

The program is translated to a simple machine independent intermediate language.

Register allocation

The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

Machine code generation

The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.

Assembly and linking

The assembly-language code is translated into binary representation and addresses of variables, functions, etc., are determined.

Structure of a Compiler (4/4)

The first three phases are collectively called the **frontend of the compiler** and

The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimizations and transformations on the intermediate code.

The last three phases are collectively called the **backend**.

Compilation – Analysis Part

- Phases involved:
 - Lexical Analysis
 - Syntax Analysis
 - Semantic Analysis
- Determines the operations implied by the source program which are recorded in a tree structure called the Syntax Tree
 - Breaks up the source code into constituent pieces, while storing information in the symbol table
 - Imposes a grammatical structure on these pieces.

Compilation – Synthesis Part

- Phases involved:
 - Intermediate Code Generation
 - (Intermediate Code Optimization)
 - Machine Code Generator
 - Machine Dependent Code Optimization
- Constructs the target code from the syntax tree, and from the information in the symbol table
- Optimization is another important activity
 - Both the intermediate and machine codes are optimized in order to achieve an efficient translation, with the least possible use of computing resources.

Front End & Back End

- **Front End phases:** (Front end is machine independent)
 - Lexical Analysis
 - Syntax Analysis
 - Semantic Analysis
- **Intermediate Code Generation**
 - (Intermediate Code Optimization)
- **Back End phases:** (Back end is machine dependent)
 - Machine-Independent Code Optimization
 - Machine Code generation
 - Machine-Dependent Code optimization

Lexical Analysis

- Read the character stream and break it up into meaningful tokens (*Scanning*)
- Each lexeme is represented by a token
 - $\langle \text{token_name}, \text{token_value} \rangle$
 - Single, atomic unit
- e.g., $\text{force} = \text{mass} * 60$
 - Lexeme force : token $\langle \text{force}, \text{force} \rangle$
 - Lexeme $=$: token $\langle \text{equal_sign}, \text{=}$
 - Lexeme mass : token $\langle \text{mass}, \text{mass} \rangle$
 - Lexeme $*$: token $\langle \text{multiplier}, \text{*}$
 - Lexeme 60 : token $\langle 60, 60 \rangle$.
- Output: $\langle \text{id}, 1 \rangle = \langle \text{id}, 2 \rangle * 60$

Formal modelling and validation of Campus Management System: Event-B perspective

Page 1 of 16
12/19/2017 12:26 PM

Nadeem Akhtar

Abstract

A Campus Management System (CMS) provides the information processing and management services that are of vital importance for the working of a department. A CMS has been formally specified, modeled, and validated for the Baghdad-ul-Jadeed campus of The Islamia University of Bahawalpur, Pakistan.

This CMS provides a formally validated correct platform for automation and management of all the major tasks of the university campus.

In order to ensure correctness of the proposed system one of the most important approach is formal specification, validation and formal proofs.

Model-based methods are based on underlying mathematical concepts of set theory and first-order predicate calculus. Among these model based methods Event-B has an exhaustive, heavy-weight, practical, efficient design and development platform named RODIN.

As a result of this high quality tool support Event-B is more practical and have far reaching industrial implementations. Event-B is centered on the key software engineering concepts of abstraction and refinement.

Keywords

Campus Management System (CMS), Formal modelling, Formal validation, Theorem Proving, First-order predicate logic, Event-B, RODIN.

1. Introduction

Universities play a major role in the higher education of a society. Science and technology plays a major important role improving the quality of life of people of a region. University education provides the required technical skills and education for the industry. The education and technical skills required for the jobs in the industry.

An information management system has been proposed for the Department of Computer Science and IT, The Islamia University of Bahawalpur, Pakistan. It is the among the two biggest public sector university in the region of South Punjab. We have approximately ten thousand students enrolled in different programs of MCS (Master in Computer Science), BS (Computer Science), BS (Information Technology), MSCS (Master of Science in Computer Science), and PhD (Computer Science) programs of the Department of Computer Science and IT.

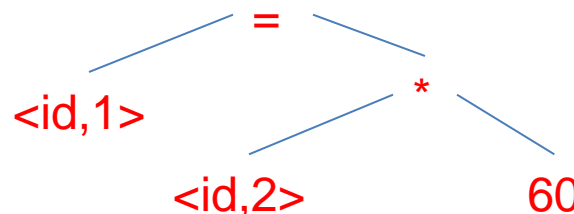
the code and break
lexemes (a.k.a

1	force	..
2	mass	..

SYMBOL TABLE

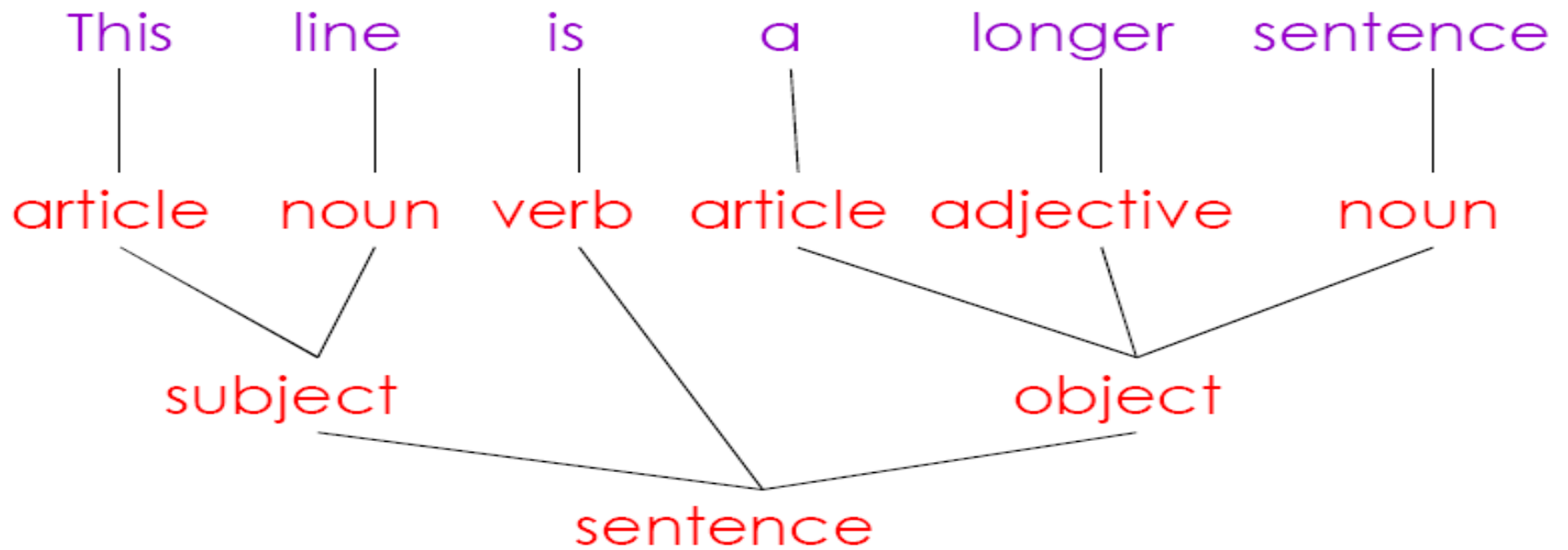
Syntax Analysis

- Syntax analysis: Parsing the token stream to identify the grammatical structure of the stream (i.e. parsing)
- Typically builds a **parse tree**, which replaces the linear sequence of tokens with a tree structure
 - The tree is built according to the rules of a formal grammar which define the language's syntax
 - It is analyzed, augmented, and transformed by later phases of compilation.

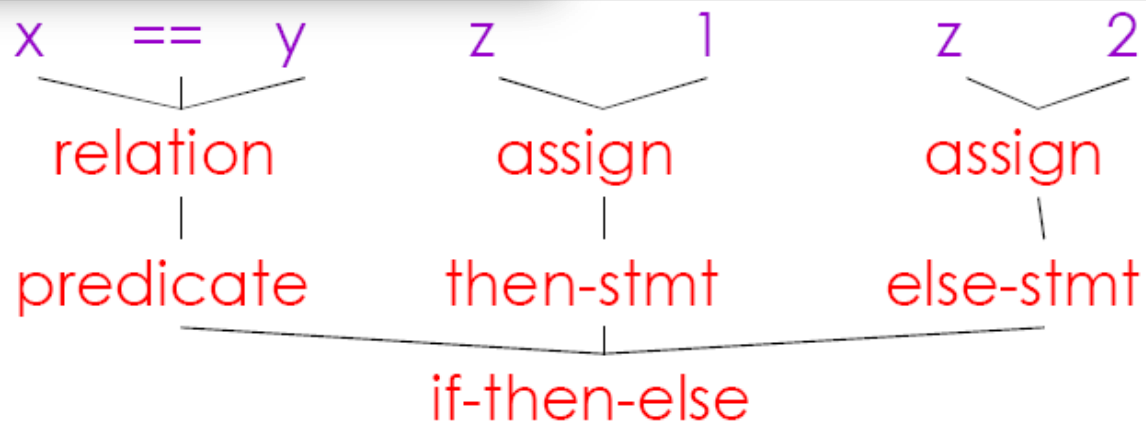


Syntax tree for
 $\langle \text{id}, 1 \rangle = \langle \text{id}, 2 \rangle * 60$

More Examples



If $x == y$ then $z = 1$; else $z = 2$;

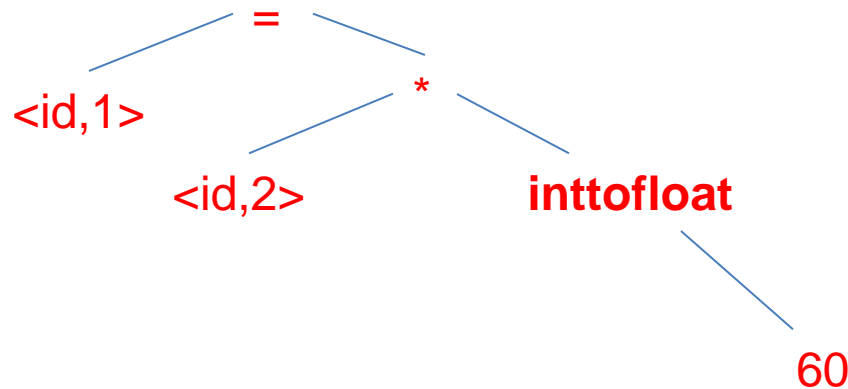


Semantic Analysis (1/2)

- Semantic analysis: Adding semantic information to the parse tree
- Performs **Semantic Checks**, e.g., **type checking** (checking for type errors), **object binding** (associating variable and function references with their definitions), **rejecting incorrect programs** or **issuing warnings**

Semantic Analysis (2/2)

- Suppose *force* and *mass* are floats and *acceleration* is integer: type casting required



Semantically correct
syntax tree for
<id,1> = <id,2> * 60

Example: English Language

- Ahmad told us that Faisal was going to his place
 - Who does *his* refer to?
- In programming language, such ambiguities are avoided

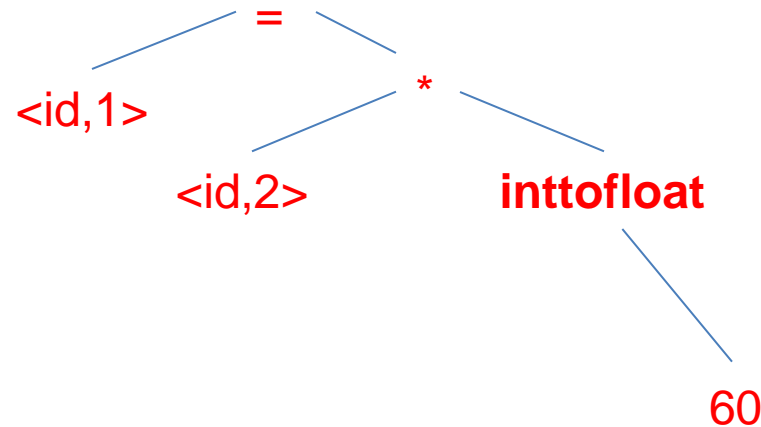
```
{  
    int Ahmad = 10;                                15 is printed  
    {  
        int Ahmad = 15;  
        System.out.println(Ahmad);  
    }  
}
```

Intermediate Code Generator (1/2)

- Intermediate Code Generation: After verifying the semantics of the source code, we convert it into a *machine-like* intermediate representation i.e., like a program for a machine
- Typically, an *assembly language-like form* is used, because it is easy to generate and debug

Intermediate Code Generator (2/2)

- **Three address code**: 3 operands/instruction
 - Suppose t1 and t2 are registers
 - `t1 = inttofloat(60)`
 - `t2 = id2*t1`
 - `id1 = t2.`



Machine-Independent Optimizer

- Machine-Independent Optimizer: Optimize the intermediate code, so that a *better* target program can be generated


- Faster, smaller code that consumes less computation power
- E.g., $X = Y * 0$ is the same as $X = 0$

- Example:

<ul style="list-style-type: none">– <code>t1 = inttofloat(60)</code>– <code>t2 = id2 * t1</code>– <code>id1 = t2</code>		<ul style="list-style-type: none"><code>t1 = id2 * 60.0</code><code>id1 = t1</code>
---	---	--

Eliminating the one time use of register t2, and converting 60 to a float

Machine Code Generation

- Machine Code Generator: Converts the optimized intermediate code into the target code (typically the Machine Code)
- This is done through the Assembly Language
- In case of Machine Code, registers (memory locations) are selected for each variable
- Then, intermediate instructions are translated into sequences of machine code instructions that perform the same task
- Example:
 - $Id1 = id2 * 60.0$  LDF R2, id2
MULF R1, R2, #60

Transfer id2 into R2, multiply
and assign to R1

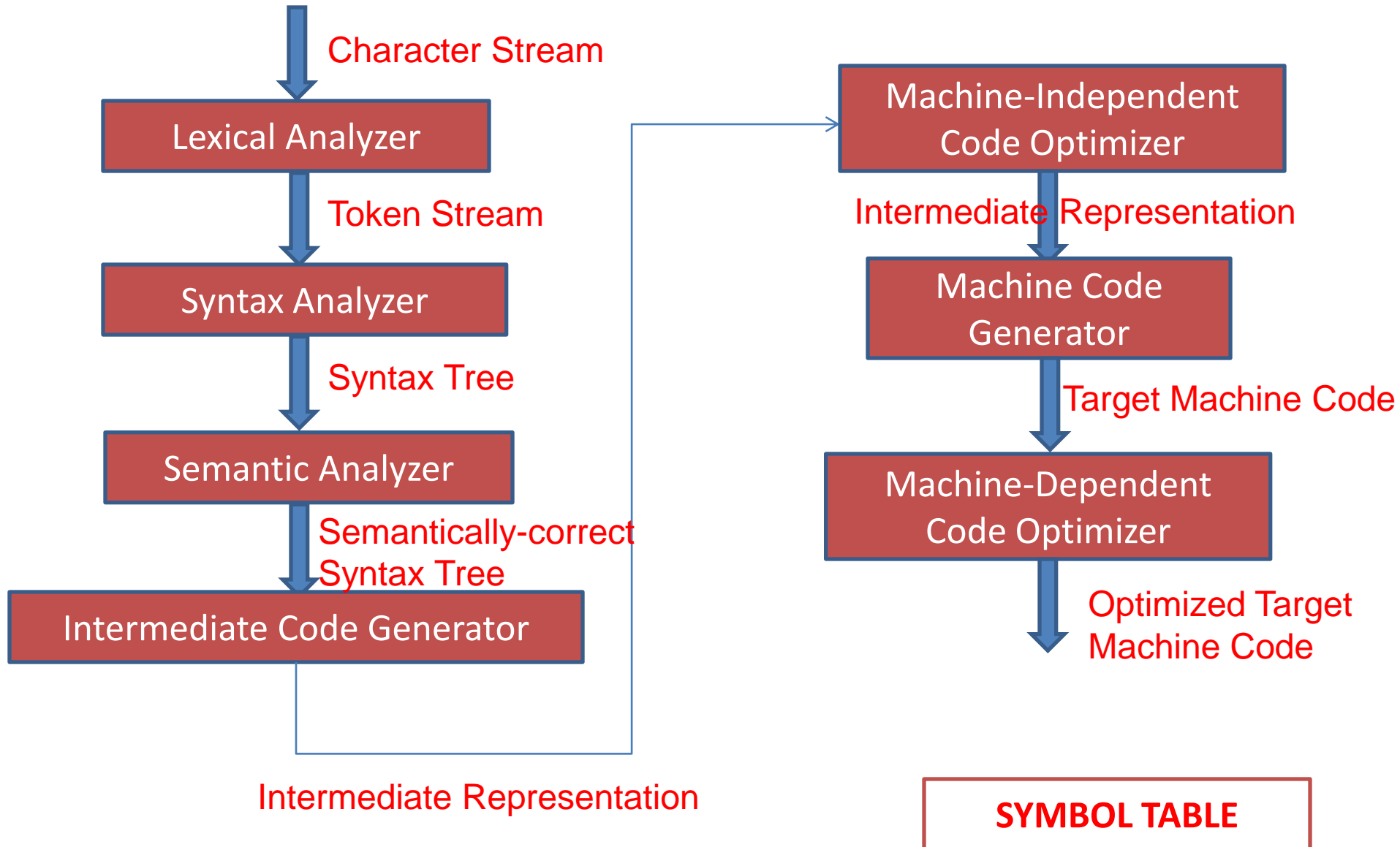
Machine-Dependent Optimizer

- Machine-Dependent Optimizer: In this phase, after the Machine Code has been generated, it is optimized further (if needed)
- This completes the compilation process
- Important: Optimization is an optional activity in compilation
 - One or both of the optimization phases might be missing.

Symbol Table Management

- **Symbol Table:** A data structure for storing the names of the variables, and their associated attributes
 - Attributes: *Storage allocated, type, scope*
 - Attributes (functions): *Number and type of arguments, method of argument passing (by reference / by value)*
- It should be designed so that the compiler can:
 - Find the record for each variable quickly
 - Store/retrieve data for a record quickly.

Structure of a Compiler



Example (1/5)

1- Lexical Analysis

- **position = initial + rate * 60**

- Lexeme position: token <id, 1>
- Lexeme =: token <=>
- Lexeme initial: token <id, 2>
- Lexeme +: token <+>
- Lexeme rate: token <id, 3>
- Lexeme *: token <*>
- Lexeme 60: token <60>

1	Position	..
2	Initial	..
3	Rate	..

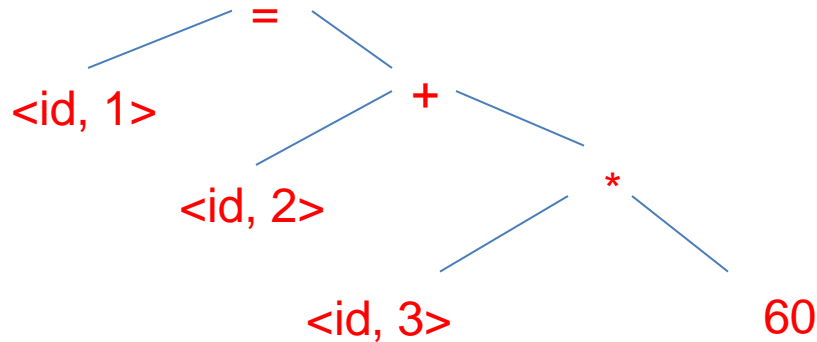
SYMBOL TABLE

- **Output: <id,1> = <id,2> <+> <id,3> <*> <60>**

Example (2/5)

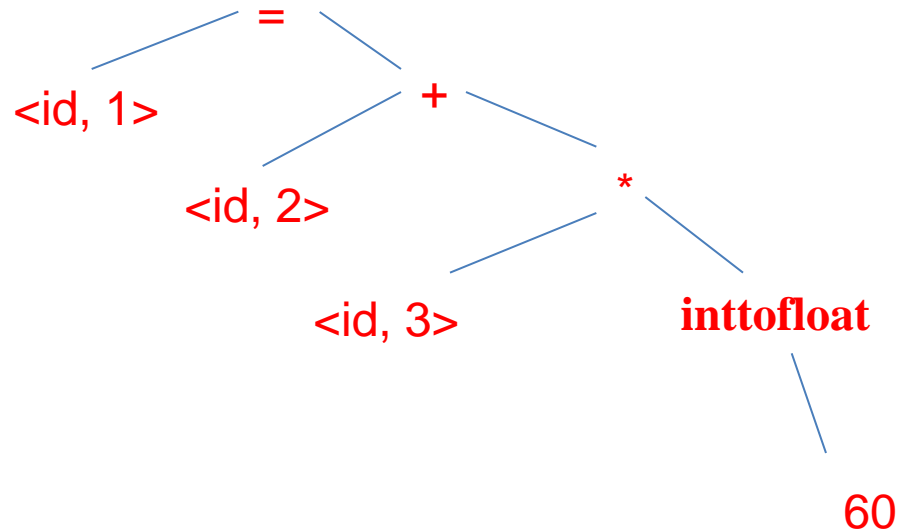
2- Syntax Analysis

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



Example (3/5)

3- Semantic Analysis



Example (4/5)

4- Intermediate Code Generation

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

5- Code optimizer

```
- t1 = id3 * 60
- id1 = id2 + t1
```

Example (5/5)

6- Code Generator

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

Grouping of Phases into Passes

- One or more phases can be grouped into a pass that reads an input file and generates an output file, e.g., *front-end phases* can be grouped into one pass
- It is also possible to skip some phase(s), e.g., all optimization activity can be avoided

Grouping of Phases into Passes

- Typically, compilers produce intermediate code from a source language that can be translated into different target languages
 - A front end can be interfaced with different back-ends.
- Several front-ends (source code in different languages) can be interfaced with the same back-end.

Compiler Construction Tools (1/3)

- Specialized tools to help implement various phases of a Compiler
- These tools use specialized languages for specifying and implementing specific components, and many use specialized algorithms.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

Compiler Construction Tools (2/3)

Sr. no.	Tool	Description
1.	Parser generators	Automatically produce syntax analyzers from a grammatical description of a programming language.
2.	Scanner generators	Produce lexical analyzers from a regular-expression description of the tokens of a language
3.	Syntax-directed translation engines	Produce collections of routines for walking a parse-tree and generating intermediate code
4.	Code-generator generators	Produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine

Compiler Construction Tools (3/3)

Sr. no.	Tool	Description
5.	Data-flow analysis engines	Facilitate the gathering of information about how values are transmitted from one part of program to each other part. Data-flow analysis is a key part of code optimization.
6.	Compiler-construction toolkits	Provides an integrated set of routines for constructing various phases of a compiler.

Programming Languages (1/10)

- First generation language: *Machine Language*
 - Very low-level operations coded in 0's and 1's
 - The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values , and so on.
 - Slow, tedious and error prone
 - Programs were hard to understand and modify

Programming Languages (2/10)

- Second generation language: *Assembly Language*
 - First step towards people-friendly language
 - Instructions are **mnemonic representations of machine instructions**
 - Macros (shorthands) are available for frequently executed pieces of code.

Programming Languages (3/10)

- Third generation languages: More user-friendly than the second-generation ones
 - FORTRAN (scientific computation)
 - COBOL (business data processing)
 - LISP (symbolic computation – Artificial Intelligence)
 - C, C++, C#, Java
- Fourth generation languages: *designed for specific applications*
 - NOMAD (for report generation), SQL (DB), Postscript (text formatting)
- Fifth generation languages: *logic and constraints-based languages*
 - Prolog, OPS5.

Programming Languages (4/10)

- Imperative languages

- Program specifies *How* a computation is to be done.
- In imperative languages there is a notion of program state and statements that change the state.
- **Examples:** C, C++, C#, and Java.

- Declarative languages

- A program specifies *What* computation is to be done.
- **Examples:** Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be declarative languages.

Programming Languages (5/10)

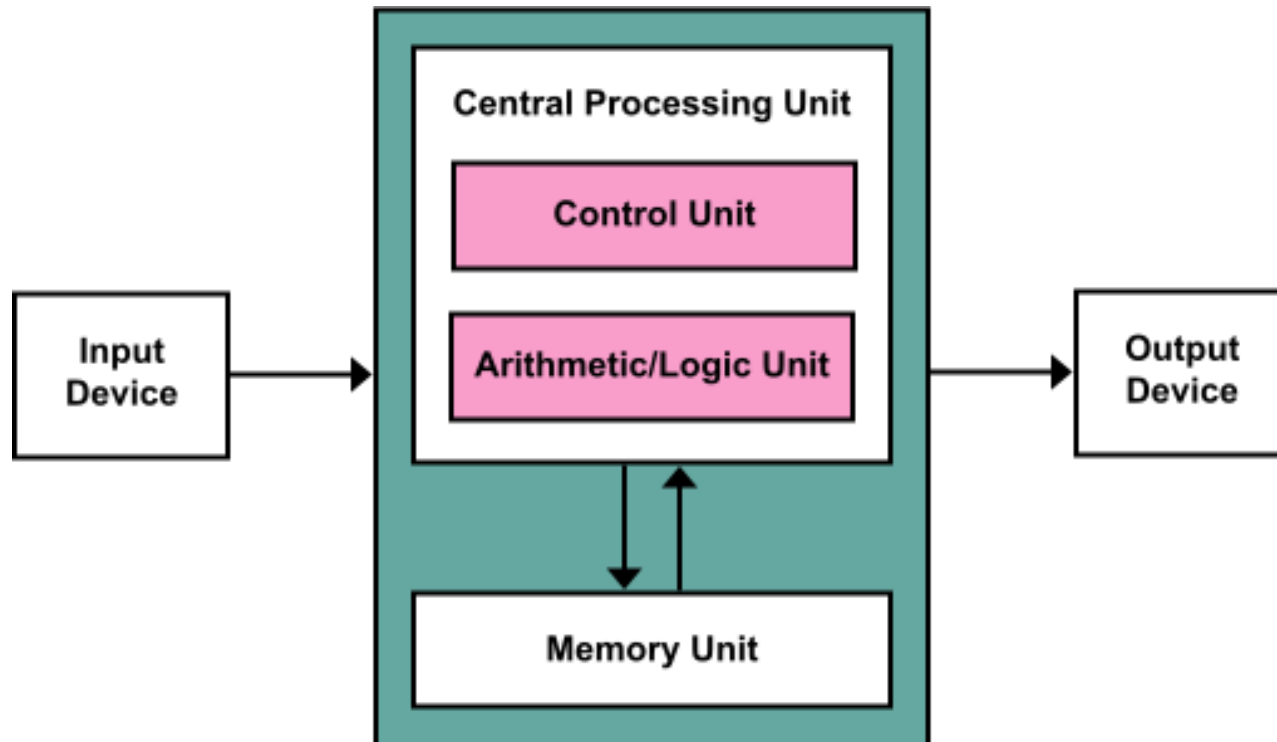
- von Neumann languages
 - The term von Neumann language is applied to programming languages whose computational model is based on the von Neumann computer architecture.
 - **Examples:** Languages, such as Fortran and C are von Neumann languages.

Programming Languages (6/10)

- **von Neumann Computer Architecture (Cambridge)**
 - Computer architecture based on that described in 1945 by John von Neumann.
 - *This describes a design architecture for an electronic digital computer with parts consisting of a processing unit containing an arithmetic logic unit and processor registers; a control unit containing an instruction register and program counter; a memory to store both data and instructions; external mass storage; and input and output mechanisms. This design architecture proposes a stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus.*

Programming Languages (7/10)

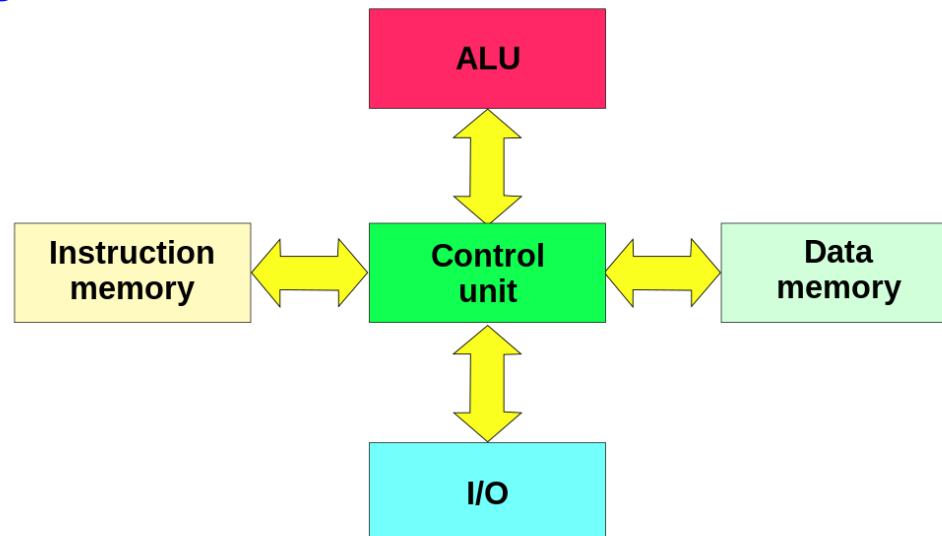
- von Neumann Computer Architecture (Cambridge)



Programming Languages (8/10)

- **Harvard Architecture**

- Computer architecture with physically separate storage and signal pathways for instructions and data.
- Has one dedicated set of address and data buses for reading data from and writing data to memory, and another set of address and data buses for fetching instructions.



Programming Languages (9/10)

- Object-oriented languages

- Supports object-oriented programming, a programming style in which a program consists of a collection of objects that interacts with one another
- **Examples:** Simula 67, SmallTalk, C++, C#, Java, Ruby etc.

- Scripting Languages

- Interpreted languages with high-level operators designed for “gluing together” computations. These computations were originally called “scripts”
- Program written in scripting languages are often much shorter than equivalent programs written in languages like C.
- **Examples:** Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl etc.

Programming Languages (10/10)

- **Impact of Language on Compilers**
 - Compilers should (and have) adapt(ed) to the evolution of languages
 - Promote the use of high-level languages by reducing the execution overhead
- **Impact of Computer Architecture on Compilers**
 - Architecture has evolved immensely, and compilers are now being used to evaluate a new architecture, before it can be marketed.

PLs Impacts on Compilers (1/3)

- Design of Programming languages and compilers are related
- With the evolution of **Programming languages**, Compiler writers have to devise algorithms and representations to translate and support the new language features.
- With the evolution of **Computer Architecture**, Compiler writers have to devise translation algorithms to take maximal advantage of the new hardware capabilities.

PLs Impacts on Compilers (2/3)

- Compilers promote the use of High Level Languages by minimizing the execution overhead of the programs written in these languages.
- Compilers are critical in making high-performance computer architectures effective on users applications.
- Performance of Computer system is dependent on Compiler technology. i.e. Compilers are used as tools in evaluating architectural concepts before a computer is built.

PLs Impacts on Compilers (3/3)

- **Compiler writing is challenging.** Many modern language-processing systems handle several source languages and target machines within the same framework i.e. collections of compilers consisting of millions of lines of code. (Good SE techniques are essential)
- Purpose of compiler is **to generate optimal target code from a source program.**
- **Compiler construction course deals with the methodology and fundamental ideas used in compiler design.**

Science of Building a Compiler

- Building a compiler is like solving a complicated, real-world mathematical problem
 - Take a problem
 - Formulate a mathematical abstraction that captures the key characteristics
 - Solve it using mathematical techniques
- Designing the right mathematical model, while balancing the need for generality and power against simplicity and efficiency
- Examples of models: *finite-state machines, regular expressions, context-free grammars.*

Code Optimization (1/2)

- To optimize is to produce better code i.e. more efficient
- **Code optimization is complex**: as Complexity of the processor architectures is continuously increasing.
- **Important**: Much data is now being parallelized. With the advent of multicore machines, all compilers must take advantage of multi-processor machines.

The use of rigorous mathematical foundation allows us to show that an optimization is correct and that it produce the desirable effect for all possible inputs.

Code Optimization (2/2)

- Optimization should:
 - Be **Correct**: preserve the original meaning of the compiled program.
 - **Improve performance** of many programs (and not only one).
 - Keep the **compilation time reasonable**.
 - Keep the **engineering effort required must be manageable**.
- **No perfect optimization technique exists**
 - Trade-offs should be made.

Applications of Compiler Technology (1/16)

1 - Implementation of High-Level Languages

- A high-level programming language defines a **programming abstraction**: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language.
- Higher-level programming languages are easier to program in, but are less efficient, that is , the target programs run more slowly.
- Programmers using a **low-level language have more control over a computation and can, in principle, produce more efficient code**. Unfortunately, lower-level programs are harder to write and - worse still - less portable, more prone to errors, and harder to maintain.

Applications of Compiler Technology (2/16)

Example:

Register keyword in early C programming language. It lets a programmer control which program variables reside in registers. This control became unnecessary as effective register-allocation techniques were developed, and most modern programs no longer use this language feature.

In fact, programs that use the register keyword may lose efficiency, because programmers often are not the best judge of very low-level matters like register allocation. The optimal choice of register allocation depends greatly on the specifics of a machine architecture. Hardwiring low-level resource-management decisions like register allocation may in fact hurt performance, especially if the program is run on machines other than the one for which it was written.

Applications of Compiler Technology (3/16)

Compiler optimizations has allowed the creation of efficient target code

- Compilation techniques have been developed to deal with all high-level functionality, e.g.
 - Object-orientation (Simula, C++, C#, and Java)
 - Data Abstraction
 - Inheritance of Properties
 - Memory management
 - Garbage collection
 - Management of data flows.

Applications of Compiler Technology (4/16)

- **Example:** Java (OOP)
 - **Type Safe** (*An object cannot be used as an object of unrelated type*)
 - All array accesses are checked to ensure that they lie **within the bounds of the array**.
 - Java has **no Pointers** and does not allow pointer arithmetic.
 - It has **built-in garbage-collection facility**
 - Java is designed to support **portable and mobile code**.
 - Programs are distributed as Java **bytecode**, which must either be interpreted or compiled into **native code** dynamically, that is, at run time. **Dynamic compilation** has also been studied in other contexts, where information is extracted dynamically at run time and used to produce better-optimized code.

Applications of Compiler Technology (5/16)

2 - Optimizations for Computer Architectures

- **Parallelism:** The instruction set of modern microprocessors uses parallelism
 - Compilation techniques have been developed that can rearrange the execution of the instructions, in order to make instruction-level parallelism more effective
 - Multiprocessors:
Programmers can write **multi-threaded code** for **multiprocessor**. or parallel code can be automatically generated by a compiler from conventional sequential programs. Such a compiler hides from the programmers the details of finding parallelism in a program, distributing the computation across the machine, and minimizing synchronization and communication among the processors.

Applications of Compiler Technology (6/16)

- **Memory hierarchy:**

- A memory hierarchy consists of several levels of storage with different speeds and sizes , with the level closest to the processor being the fastest but smallest.
- Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond.
- The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem.
- While Compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

Applications of Compiler Technology (7/16)

- Compilation techniques have been developed to improve the effectiveness of the memory hierarchy by changing the data's layout, or the instruction execution order.
- Using *Registers* effectively is probably the single most important problem in optimizing a program. Unlike registers that have to be managed explicitly in software, caches and physical memories are hidden from the instruction set and are managed by hardware.

Applications of Compiler Technology (8/16)

3- Design of New Computer Architectures

- *Programming in high-level languages*, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features.

Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features

Applications of Compiler Technology (9/16)

- RISC (Reduced Instruction-Set Computer) architecture.
 - Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept.
- CISC (Complex Instruction-Set Computer) architecture
 - CISC instruction sets include complex memory-addressing modes to support data-structure accesses and procedure-invocation instructions that save registers and pass parameters on the stack.
 - The x86 architecture - the most popular microprocessor-has a CISC instruction set

Applications of Compiler Technology (10/16)

- Specialized Architectures

- They include data flow machines, vector machines,
- VLIW (Very Long Instruction Word) machines
- SIMD (Single Instruction, Multiple Data) arrays of processors,
- systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory.
- The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Applications of Compiler Technology (11/16)

4- Program Translations

Compiler technology to translate between different kinds of languages.

- Binary Translation

- Binary translators have been developed to convert x86 code into both Alpha and Sparc code. Binary translation was also used by Transmeta Inc. in their implementation of the x86 instruction set. Instead of executing the complex x86 instruction set directly in hardware, the Transmeta Crusoe processor is a VLIW processor that relies on binary translation to convert x86 code into native VLIW code.
- Binary translation can also be used to provide backward compatibility.

Applications of Compiler Technology (12/16)

- Hardware Synthesis

High-level hardware description languages like Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language)

Hardware designs are typically described at the Register Transfer Level (RTL) , where variables represent **Registers** and expressions represent **Combinational logic**.

Hardware-synthesis tools translate RTL descriptions automatically into gates, which are then mapped to transistors and eventually to a physical layout.

Applications of Compiler Technology (13/16)

- Database Query Interpreters

Query languages, especially **SQL (Structured Query Language)**, are used to search databases.

Database queries consist of predicates containing relational and Boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate.

Applications of Compiler Technology (14/16)

- **Compiled Simulation**

Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive.

A single design simulation involves many possible design alternatives on many different input sets, and each experiment may take days to complete on a high-performance machine.

Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the-art tools that simulate designs written in Verilog or VHDL.

Applications of Compiler Technology (15/16)

4 – Software Productivity Tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems.

Testing is the primary technique for locating errors in programs.

An interesting and promising complementary approach is to use **data-flow analysis** to locate errors statically (that is, before the program is run). Dataflow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing.

Applications of Compiler Technology (16/16)

- Type Checking
- Bounds Checking
- Memory Management Tools
 - Garbage Collection
 - Automatic memory management removes all memory-management errors (e.g. , "memory leaks"), which are a major source of problems in C and C++ programs.

Various tools have been developed to help programmers find memory management errors.

Purify is a widely used tool that dynamically catches memory management errors as they occur. Tools that help identify some of these problems statically have also been developed.

Programming Basics

- Building a compiler depends on what decisions the compiler can take about the given program
 - **Static Policy:** If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at *compile time*.
 - **Dynamic Policy:** A policy that only allows a decision to be made when we execute the program is said to be dynamic policy or to require decision at run time.

The Static/Dynamic Distinction (1/3)

- Static Scope or Lexical Scope
 - The scope of a declaration of *x* is the region of the program in which uses of *x* refer to this declaration.
 - A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the code (i.e. program before exec)
 - The keyword “static” in Java: `static int x`
 - The compiler can determine the location of *x* (in memory)
 - Use of **public**, **protected** and **private** keywords.

The Static/Dynamic Distinction (2/3)

- **Dynamic Scope**

- When the scope can only be determined at runtime. With the dynamic scope the same use of x could refer to any of several different declarations of x .
- The value of variable can be computed at run time, e.g., based on the input of the user

Most languages, such as C and Java, use static scope.

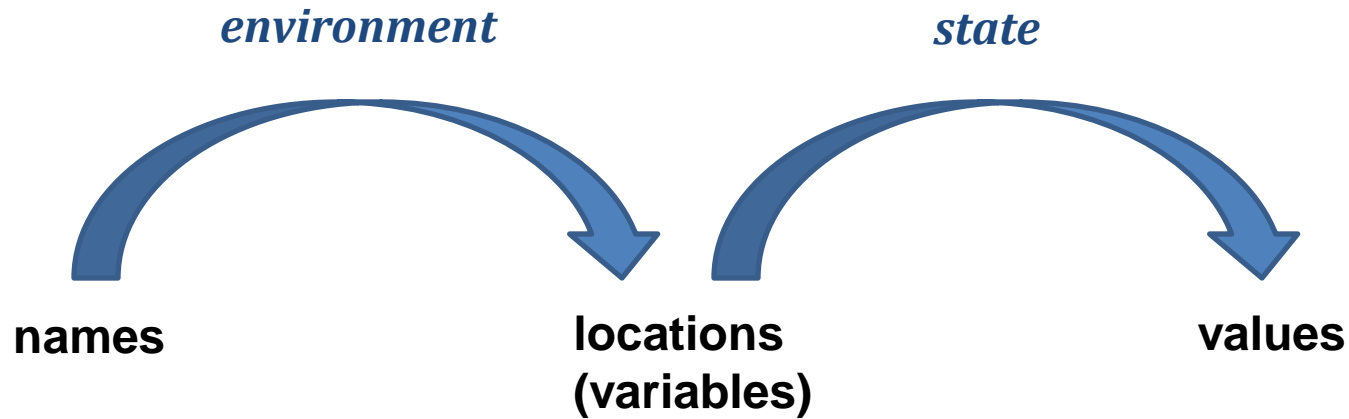
The Static/Dynamic Distinction (3/3)

- Example

- `public static int x;`

- Here `x` is a ***class variable*** and says that there is only one copy of `x`, no matter how many objects of this class are created.
 - Moreover, the compiler can determine the a location in memory where this integer `x` will be held.

Environments and States (1/4)



Two-stage mapping from names to values

Environments and States (2/4)

The *environment* is a mapping from names to locations in the store. Since variables refer to locations (l-values in C). Environment as a mapping from names to variables.

The *state* is a mapping from location in store to their values. State maps l-values to their corresponding r-values (i.e. C).

Environments and States (3/4)

Example

```
...  
int i;    /* global i */  
...  
void f(...) {  
    int i;    /* local i */  
    ...  
    i = 3;    /* use of local i */  
    ...  
}  
...  
x = i + 1;    /* use of global i */
```

Two declarations of the name i

Environments and States (4/4)

Static versus dynamic binding of names to locations. Most binding of names to locations is dynamic.

Static versus dynamic binding of locations to values. The binding of locations to values is generally dynamic as well, since we cannot tell the value in a location until we run a program. Declared constants are exceptions. i.e. the C definition

#define ARRAYSIZE 1000

Binds the name ARRAYSIZE to the value 1000 statically.

Static Scope and Block Structure (1/7)

- The scope rules for C are based on program structure; the scope of a declaration is determined **implicitly** by where the declaration appears in the program.
- Languages C++, Java, and C#, provide **explicit** control over scopes through the use of keywords like **public**, **private**, and **protected**.

Static Scope and Block Structure (2/7)

- A **block** is a grouping of declarations and statements. C uses braces { and } to delimit a block;
- **Algol** use begin and end for the same purpose.

Static Scope and Block Structure (3/7)

In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statements, such as assignment statements, can appear.
2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

This syntax allows blocks to be nested inside each other. This nesting property is referred to as block structure. The C family of languages has block structure, except that a function may not be defined inside another function.

Static Scope and Block Structure (4/7)

The C static-scope policy is as follows:

1. A C program consists of a sequence of **top-level declarations of variables and functions**.
2. **Functions** may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.
3. The scope of a top-level declaration of a name *x* consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of *x*.

Static Scope and Block Structure (5/7)

The static-scope rule for variable declarations in a block-structured languages is as follows:

If declaration D of name x belongs to block B , then the scope of D is all of B , except for any blocks B' nested to any depth within B , in which x is redeclared. Here, x is redeclared in B' if some other declaration D' of the same name x belongs to B' .

Static Scope and Block Structure (6/7)

```
main() {  
    int a = 1;  
    int b = 1;  
    {  
        int b = 2;  
        {  
            int a = 3;  
            cout << a << b;  
        }  
        {  
            int b = 4;  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

The diagram illustrates the static scope and block structure of a C++ program. It shows a main function containing several nested blocks. The blocks are labeled B_1 , B_2 , B_3 , and B_4 . B_1 is the outermost block, containing the main function's body. B_2 is a block nested within B_1 . B_3 and B_4 are blocks nested within B_2 . The code is as follows:

```
main() {  
    int a = 1;  
    int b = 1;  
    {  
        int b = 2;  
        {  
            int a = 3;  
            cout << a << b;  
        }  
        {  
            int b = 4;  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

Blocks in a C++ Program

Static Scope and Block Structure (7/7)

DECLARATION	SCOPE
<code>int a = 1;</code>	$B_1 - B_3$
<code>int b = 1;</code>	$B_1 - B_2$
<code>int b = 2;</code>	$B_2 - B_4$
<code>int a = 3;</code>	B_3
<code>int b = 4;</code>	B_4

Scope of Declarations

Procedures, Functions, and Methods

- A **function** generally returns a value of some type (the "return type"), while a procedure does not return any value.
 - C has only **functions**, treat procedures as functions that have a special return type "void," to signify no return value.
 - Object-oriented languages like Java and C++ use the term "**methods**". *These can behave like either functions or procedures, but are associated with a particular class.*

Explicit Access Control (1/3)

- The use of keywords like *public*, *private*, and *protected*
- Classes and structures introduce a new scope for their members. If p is an object of a class with a field (member) x , then the use of x in $p.x$ refers to field x in the class definition. The scope of a member declaration x in a class C extends to any subclass C' , except if C' has a local declaration of the same name x .

Explicit Access Control (2/3)

- Through the use of keywords like *public*, *private*, and *protected*, object oriented languages (i.e. C++, Java) provide explicit control over access to member names in a superclass.
- These keywords support encapsulation by restricting access.

Explicit Access Control (3/3)

- **Private** names have a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C++ term).
- **Protected** names are accessible to subclasses.
- **Public** names are accessible from outside the class.

Dynamic Scope (1/5)

- A scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes.
- The term dynamic scope, however, refers to the following policy:
 - *a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration.*

Dynamic Scope (2/5)

- **Example:** *(macro expansion in the C preprocessor)*

```
#define a (x+1)

int x = 2;

void b() { int x = 1; printf("%d\n", a); }

void c() { printf("%d\n", a); }

void main() { b(); c(); }
```

A macro whose names must be scoped dynamically

Dynamic Scope (3/5)

- Dynamic scope resolution is also essential for **polymorphic procedures**, those that have two or more definitions for the same name, depending only on the types of the arguments.

Example - Dynamic Scope (4/5)

- **Example:** *(method resolution in object-oriented programming)*
- *A distinguishing feature of object-oriented programming is the ability of each object to invoke the appropriate method in response to a message.*
- *In other words, the procedure called when $x.m()$ is executed depends on the class of the object denoted by x at that time.*

Example - Dynamic Scope (5/5)

A typical example is as follows:

- 1. There is a class C with a method named m().*
- 2. D is a subclass of C, and D has its own method named m().*
- 3. There is a use of m of the form x.m() , where x is an object of class C.*

Normally, it is impossible to tell at compile time whether x will be of class C or of the subclass D . If the method application occurs several times , it is highly likely that some will be on objects denoted by x that are in class C but not D , while others will be in class D . It is not until run-time that it can be decided which definition of m is the right one . Thus , the code generated by the compiler must determine the class of the object x, and call one or the other method named m.

Parameter Passing Mechanisms

The *actual parameters* (the parameters used in the call of a procedure)

The *formal parameters* (the parameters used in the procedure definition)

- Call by Value
- Call by Reference
- Call by Name

Call by Value (1/5)

- In **call-by-value**, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable) . The value is placed in the location belonging to the corresponding formal parameter of the called procedure.

Example: C, C++, and Java

- In **call-by-value** all computation involving the formal parameters done by the *called procedure* is *local to that procedure*, and the *actual parameters* themselves cannot be changed.

Call by Value (2/5)

- In C we can pass a pointer to a variable to allow that variable to be changed by the callee.
- *In C, C++, and Java array names are passed as parameters.*

Call by Value (3/5)

- Java give the *called procedure* what is in effect *a pointer or reference to the array itself*.
- Thus, if *a* is the name of an array of the calling procedure, and it is passed by value to corresponding formal parameter *x*, then an assignment such as *x[i]=2* really changes the array element *a[2]*.
- *Although *x* gets a copy of the value of *a*, that value is really a pointer to the beginning of the area of the store where the array named *a* is located.*

Call by Value (4/5)

- *Java*, many variables are really references, or pointers, to the things they stand for. This observation applies to arrays , strings, and objects of all classes.
- Java uses call-by-value exclusively, whenever the name of an object is passed to a called procedure, the value received by that procedure is in effect a pointer to the object. *Thus, the called procedure is able to affect the value of the object itself.*

Call by Value (5/5)

- When large objects are passed by value (i.e. formal parameter is *a large object , array, or structure*), the values passed are really references to the objects themselves, resulting in an effective call-by-reference.
 - Reason: *strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large.*

Call by Reference (1/3)

- In call-by-reference, the address of the actual parameter is passed to the *callee* as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the *callee* are implemented by following this pointer to the location indicated by the caller.
- Changes to the formal parameter thus appear as changes to the actual parameter.

Call by Reference (2/3)

- *If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.*

Call by Reference (3/3)

- *Call by reference is almost essential when the formal parameter is a large object , array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large.*
- *It is to be noted that in call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.*

Call by Name

- *A third mechanism - call-by-name - was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct).*
- *When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.*

Summary

- **Language Processor.** An integrated software development environment includes many different kinds of language processors such as *compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.*
- **Compiler Phases.** A compiler operates as a sequence of phases, each of which transforms the source program from one intermediate representation to another.

Summary

- ***Machine and Assembly Languages.*** Machine languages were the first generation programming languages, followed by assembly languages. Programming in these languages was time consuming and error prone .
- ***Modeling in Compiler Design.*** Compiler design is one of the places where theory has had the most impact on practice. Models that have been found useful include *automata*, *grammars*, *regular expressions*, *trees*, and many others.

Summary

- **Code Optimization.** Although code cannot truly be "optimized," the science of improving the efficiency of code is both complex and very important. It is a major portion of the study of compilation.
- **Higher-Level Languages.** As time goes on, programming languages take on progressively more of the tasks that formerly were left to the programmer, such as *memory management, type-consistency checking, or parallel execution of code.*

Summary

- ***Compilers and Computer Architecture.*** Compiler technology influences computer architecture, as well as being influenced by the advances in architecture. Many modern innovations in architecture depend on compilers being able to extract from source programs the opportunities to use the hardware capabilities effectively.
- ***Software Productivity and Software Security.*** The same technology that allows compilers to optimize code can be used for a variety of program analysis tasks, ranging from detecting common program bugs to discovering that a program is vulnerable to one of the many kinds of intrusions that "hackers" have discovered.

Summary

- **Scope Rules.** The scope of a declaration of ***x*** is the context in which uses of ***x*** refer to this declaration. A language uses ***static scope*** or ***lexical scope*** if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses dynamic scope.
- **Environments.** The association of ***names*** with ***locations*** in memory and then with ***values*** can be described in terms of *environments, which map names to locations in store*, and *states, which map locations to their values*.

Summary

- **Block Structure.** Languages that allow blocks to be nested are said to have block structure. A name x in a nested block B is in the scope of a declaration D of x in an enclosing block if there is no other declaration of x in an intervening block.
- **Parameter Passing.** Parameters are passed from a calling procedure to the callee either **by value** or **by reference**. *When large objects are passed by value, the values passed are really references to the objects themselves, resulting in an effective call-by-reference.*

Summary

- ***Aliasing.*** When parameters are (effectively) passed by reference, *two formal parameters can refer to the same object*. This possibility allows a change in one variable to change another.

Compiler Construction with Java (1/2)

- **ANTLR: Another Tool for Language Recognition**
 - Provides scanning, parsing and tree-walking functionalities, along with many others
 - <http://www.antlr.org/>
- **JACCIE: Java-based Compiler Compiler with Interactive Environment**
 - Educational tool for visualizing modern compiling techniques
 - Automatically generates demonstration compilers from suitable language descriptions
 - <http://www2.cs.unibw.de/Tools/Syntax/english/index.html>

Compiler Construction with Java (2/2)

- **SableCC**: A compiler construction toolkit
 - <http://sablecc.org/>
- **JavaCC**: Very popular parser generator
 - Takes as input a language, converts it to a Java file, and then determines matches to this language
 - Builds syntax trees according to a given language using **Java Tree Builder (JTB)**
 - <https://javacc.dev.java.net/>
- **JLEX**: A Java-based Lexical analyzer
 - <http://www.cs.princeton.edu/~appel/modern/java/JLex/>

With C# - IRONY

- IRONY: A development kit for implementing languages on .NET platform
 - Uses the flexibility and power of C# language and .NET Framework to implement a completely new and streamlined technology of compiler construction
 - Provides a new programming approach to encode some grammar
 - <http://www.codeproject.com/KB/recipes/Irony.aspx>
 - <http://www.codeplex.com/irony>

With C# -Z# Compiler

- **Compiler Construction Course: Concepts and Practical Applications to .NET**
- Wades through all phases of a compiler
- Shows theoretical concepts and implementation details for each phase
- Task is to write a small compiler for a C#-like programming language (Z#)
 - Translate a source program into the code of a virtual machine (CLR)
- <http://dotnet.jku.at/courses/CC/>

PHC & YACC

- PHC: open source compiler for PHP with support for plugins
 - Not as flexible as libraries for Java/C#
 - <http://www.phpcompiler.org/>
- While browsing, you will come across “YACC”
 - YACC: Yet Another Compiler Compiler
 - A parser generator for UNIX that outputs in C
 - The default parser generator for UNIX for a long time
 - Now replaced by Berkeley YACC, GNU Bison, MKS Yacc etc.
 - Many C-based CC tools require YACC-like inputs.