

Introduction to Compiler Construction

Robert van Engelen

<http://www.cs.fsu.edu/~engelen/courses/COP5621>

Syllabus

- Prerequisites: COP4020 or equivalent
- Textbook: “*Compilers: Principles, Techniques, and Tools*” by Aho, Sethi, and Ullman, 2nd edition
- Other material: “*The JavaTM Virtual Machine Specification*”, 2nd edition and class handouts
- Grade breakdown:
 - Four exams (60%)
 - Four programming assignments (40%)
 - Homework for extra credit (up to 4%)

Syllabus, Assignments, and Schedule

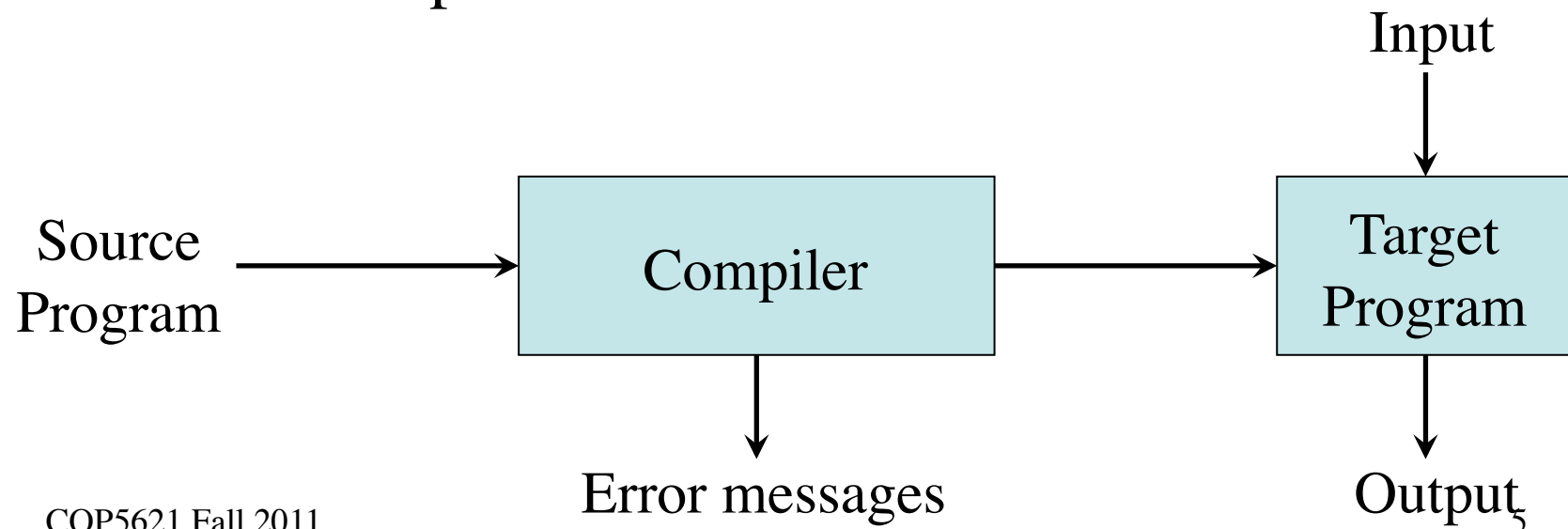
<http://www.cs.fsu.edu/~engelen/courses/COP5621>

Objectives

- Be able to build a compiler for a (simplified) (programming) language
- Know how to use compiler construction tools, such as generators of scanners and parsers
- Be familiar with assembly code and virtual machines, such as the JVM, and bytecode
- Be able to define LL(1), LR(1), and LALR(1) grammars
- Be familiar with compiler analysis and optimization techniques
- ... learn how to work on a larger software project!

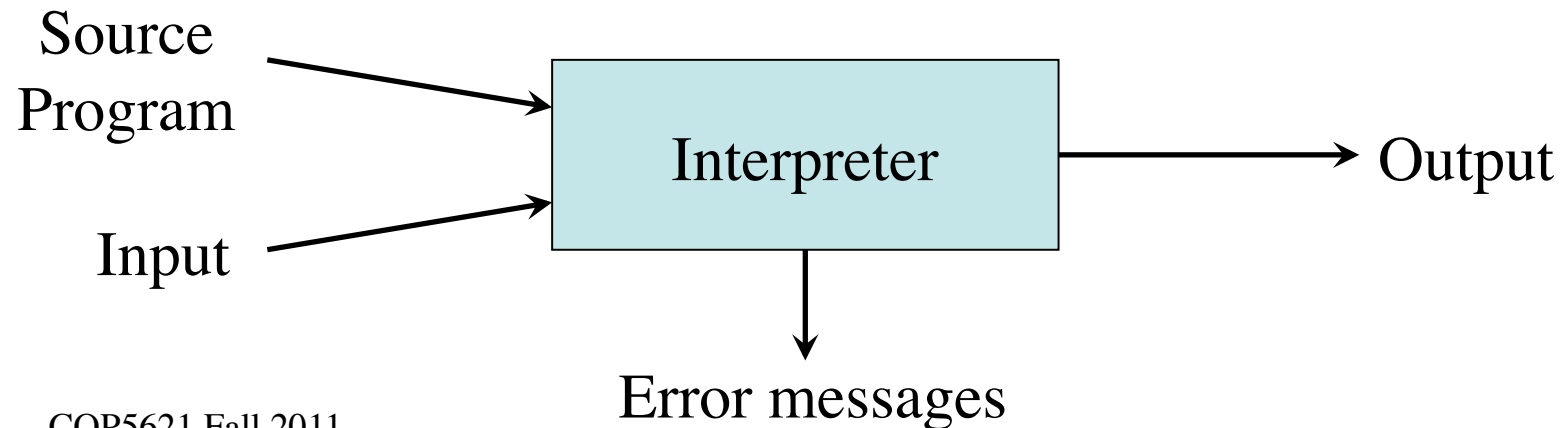
Compilers and Interpreters

- “*Compilation*”
 - Translation of a program written in a source language into a semantically equivalent program written in a target language
 - Oversimplified view:



Compilers and Interpreters (cont'd)

- “*Interpretation*”
 - Performing the operations implied by the source program
 - Oversimplified view:



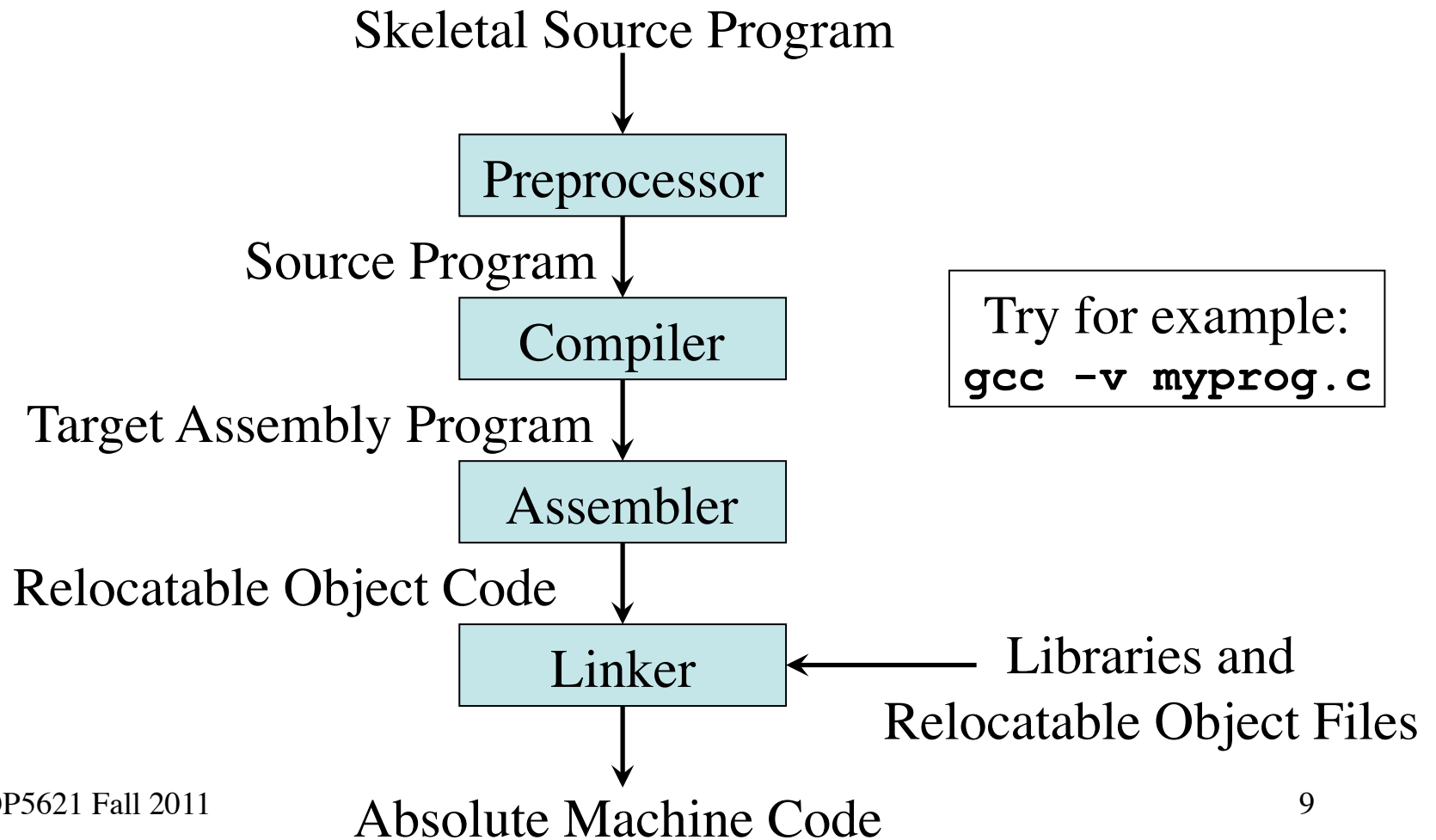
The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
 - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
 - *Synthesis* takes the tree structure and translates the operations therein into the target program

Other Tools that Use the Analysis-Synthesis Model

- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. Doxygen)
- *Static checkers* (e.g. Lint and Splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

Preprocessors, Compilers, Assemblers, and Linkers



The Phases of a Compiler

Phase	Output	Sample
<i>Programmer (source code producer)</i>	Source string	A=B+C ;
<i>Scanner (performs lexical analysis)</i>	Token string	\A', \=', \B', \+', \C', \;' And <i>symbol table</i> with names
<i>Parser (performs syntax analysis based on the grammar of the programming language)</i>	Parse tree or abstract syntax tree	<pre> ; = / \ A + / \ B C </pre>
<i>Semantic analyzer (type checking, etc)</i>	Annotated parse tree or abstract syntax tree	
<i>Intermediate code generator</i>	Three-address code, quads, or RTL	<pre> int2fp B t1 + t1 C t2 := t2 A </pre>
<i>Optimizer</i>	Three-address code, quads, or RTL	<pre> int2fp B t1 + t1 #2.3 A </pre>
<i>Code generator</i>	Assembly code	<pre> MOVFB #2.3, r1 ADDF2 r1, r2 MOVFB r2, A </pre>
<i>Peephole optimizer</i>	Assembly code	<pre> ADDF2 #2.3, r2 MOVFB r2, A </pre>

The Grouping of Phases

- Compiler *front* and *back ends*:
 - Front end: *analysis (machine independent)*
 - Back end: *synthesis (machine dependent)*
- Compiler *passes*:
 - A collection of phases is done only once (*single pass*) or multiple times (*multi pass*)
 - Single pass: usually requires everything to be defined before being used in source program
 - Multi pass: compiler may have to keep entire program representation in memory

Compiler-Construction Tools

- Software development tools are available to implement one or more compiler phases
 - *Scanner generators*
 - *Parser generators*
 - *Syntax-directed translation engines*
 - *Automatic code generators*
 - *Data-flow engines*

Outline

- Introduction
- A simple One-Pass Compiler for the JVM
- Lexical Analysis and Lex/Flex
- Syntax Analysis and Yacc/Bison
- Syntax-Directed Translation
- Static Semantics and Type Checking
- Run-Time Environments
- Intermediate Code Generation
- Target Code Generation
- Code Optimization