Translator Design

Introduction

Textbook and Other Classroom Material

- Class textbook
  - *Compilers: Principles, Techniques*, *and Tools*, Aho, Sethi, Ullman (red dragon book)
  - *Crafting a Compiler with C*,
- Other useful books
  - *Lex & Yacc*, Levine, Mason and Brown
  - *Advanced Compiler Design & Implementation*, Steven Muchnick

Course Grading

- Components
  - Exam – 60%
  - Laboratory homeworks – 40%

Why Compilers?

- Compiler
  - A program that translates from 1 language to another
  - It must preserve semantics of the source
  - It should create an efficient version of the target language

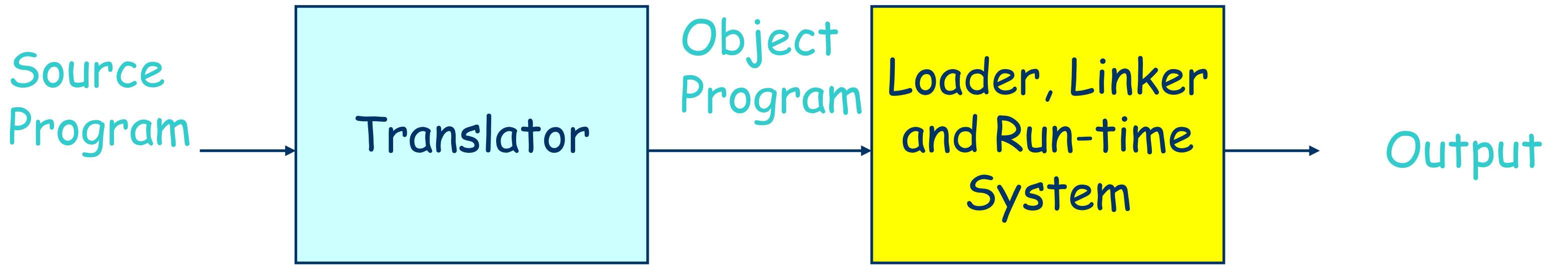
Why Compilers?

- In the beginning, there was machine language
  - Ugly – writing code, debugging
  - Then came textual assembly – still used
  - High-level languages – Fortran, Pascal, C, C++
  - Machine structures became too complex and software management too difficult to continue with low-level languages

Compilers are Translators

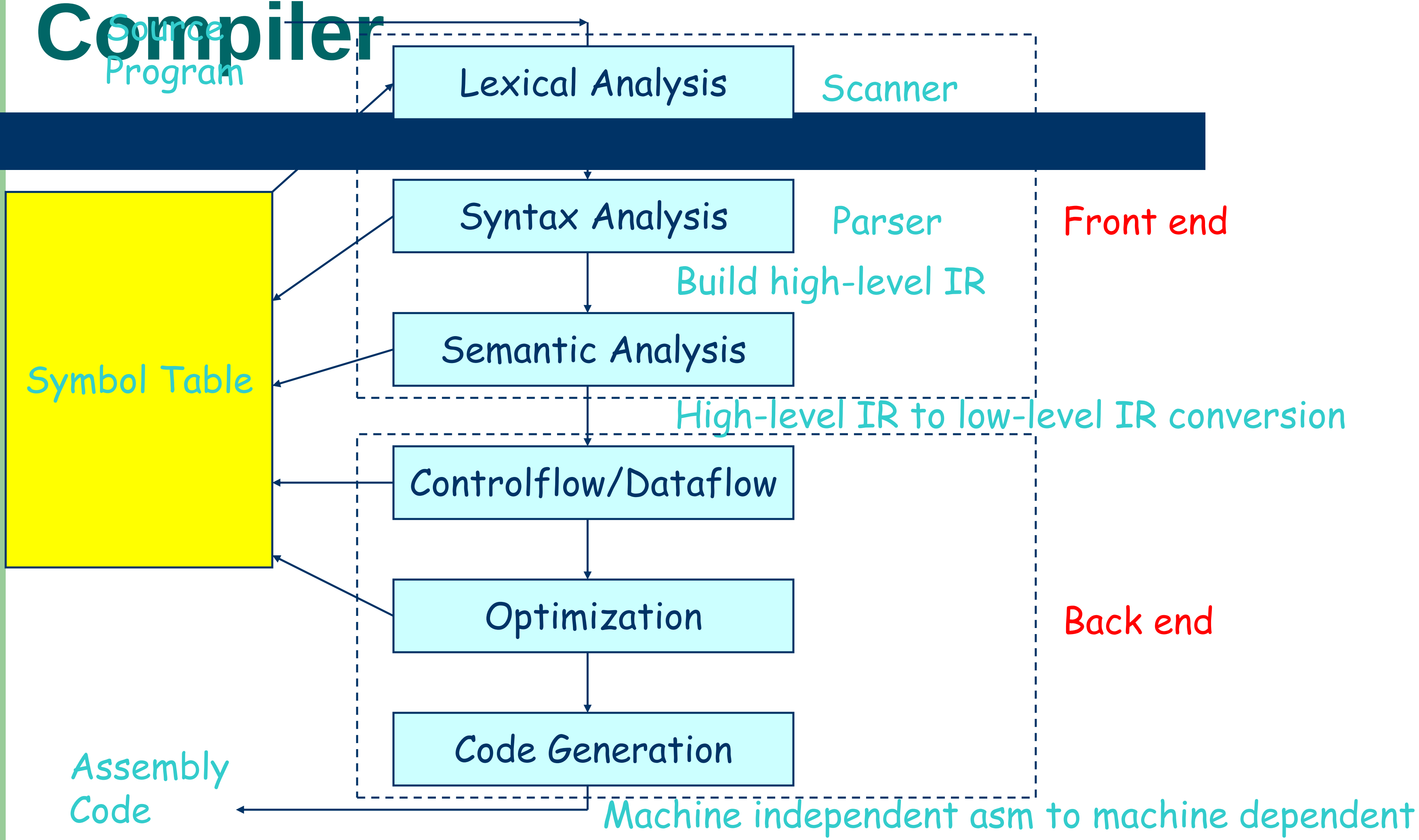
- Fortran, C, C++, Java
- Text processing language
- Command Language
- Natural language

Compiler Structure

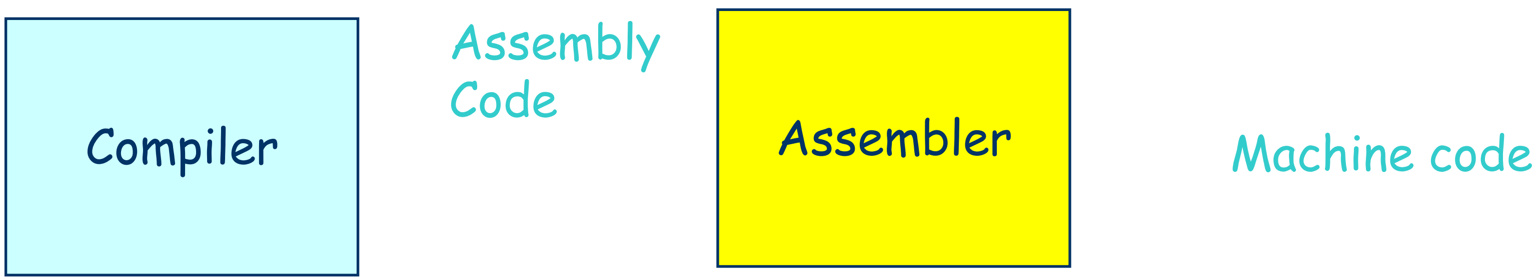


- Source language
  - Fortran, Pascal, C, C++
  - VHDL, Tex, Html
- Target language
  - Machine code, assembly
  - High-level languages, simply actions

General Structure of a Modern Compiler



Assembly code and Assemblers



- **Assemblers are often used at the compiler back-end.**
  - Assemblers are low-level translators.
  - They are machine-specific,
  - perform mostly 1:1 translation between mnemonics and machine code

Interpreters

- "Execute" the source language directly.
- Interpreters directly produce the result of a computation, whereas compilers produce executable code that can produce this result.
- Each language construct executes by invoking a subroutine of the interpreter, rather than a machine instruction.
- “execution” is immediate
- elaborate error checking is possible
- Disadvantage: is slow; space overhead

Lexical Analysis (Scanner)

- Extracts and identifies lowest level lexical elements from a source stream
  - Reserved words: for, if, switch
  - Identifiers: “i”, “j”, “table”
  - Constants: 3.14159, 17, “%d\n”
  - Punctuation symbols: “(“, “)”, “,”, “+”
- Removes non-grammatical elements from the stream – ie spaces, comments

Lexical Analysis (Scanner)

- Implemented with a Finite State Automata (FSA)
  - Set of states – partial inputs
  - Transition functions to move between states

Lex/Flex

- Automatic generation of scanners
  - Hand-coded ones are faster
  - But tedious to write, and error prone!
- Lex/Flex
  - Given a specification of regular expressions
  - Generate a table driven FSA
  - Output is a C program that you compile to produce your scanner

Parser

- Check input stream for syntactic correctness
  - Framework for subsequent semantic processing
  - Implemented as a push down automaton (PDA)
- Lots of variations
  - Hand coded, recursive descent?
  - Table driven (top-down or bottom-up)
  - For any non-trivial language, writing a correct parser is a challenge

Parser

- Yacc (yet another compiler compiler)/bison
  - Given a context free grammar
  - Generate a parser for that language (again a C program)

Static Semantic Analysis

- Several distinct actions to perform
  - Check definition of identifiers, ascertain that the usage is correct
  - Disambiguate overloaded operators
  - Translate from source to IR (intermediate representation)

Static Semantic Analysis

- Standard formalism used to define the application of semantic rules is the Attribute Grammar (AG)
  - Graph that provides for the migration of information around the parse tree
  - Functions to apply to each node in the tree

Backend

- Frontend –
  - Statements, loops, etc
  - These broken down into multiple assembly statements
- Machine independent assembly code
  - 3-address code, RTL
  - Infinite virtual registers, infinite resources
  - “Standard” opcode repetoire
    - load/store architecture

Backend

- Goals
  - Optimize code quality
  - Map application to real hardware

Dataflow and Control Flow Analysis

- Provide the necessary information about variable usage and execution behavior to determine when a transformation is legal/illegal
- Dataflow analysis
  - Identify when variables contain “interesting” values
  - Which instructions created values or consume values
  - DEF, USE, GEN, KILL

Dataflow and Control Flow Analysis

- Control flow analysis
  - Execution behavior caused by control statements
  - If’s, for/while loops, goto’s
  - Control flow graph

Optimization

- How to make the code go faster
- Classical optimizations
  - Dead code elimination – remove useless code
  - Common subexpression elimimation – recomputing the same thing multiple times
- Machine independent (classical)
  - Focus of this class
  - Useful for almost all architectures
- Machine dependent
  - Depends on processor architecture
  - Memory system, branches, dependences

Code Generation

- Mapping machine independent assembly code to the target architecture
- Virtual to physical binding
  - Instruction selection – best machine opcodes to implement generic opcodes
  - Register allocation – infinite virtual registers to N physical registers
  - Scheduling – binding to resources (ie adder1)
  - Assembly emission
- Machine assembly is our output, assembler, linker take over to create binary

Compiler Writing Tools

- Other terms: compiler generators, compiler compilers
- scanner generators, example: lex
- parser generators, example: yacc
- symbol table routines,
- code generation aids,
- (optimizer generators, still a research topic)
- These tools are useful, but bulk of work for
- compiler writer is in semantic routines and optimizations .

Sequence of Compiler Passes

- **In general, all compiler passes are run in sequence.**
  - They read the internal program representation,
  - process the information, and
  - generate the output representation.

Sequence of Compiler Passes

- **For a simple compiler, we can make a few simplifications. For example:**
  - Semantic routines and code generator are combined
  - There is no optimizer
  - All passes may be combined into one. That is, the compiler
    performs all steps in one run.
- One-pass compilers do not need an internal representation. They process a syntactic unit at a time, performing all steps from scanning to code generation.

Language Syntax and Semantics

- **An important distinction**:
  - Syntax defines the structure of a language

    E.g., an IF clause has the structure:

    IF (*expression*) THEN *statements*
  - Semantics defines its meaning

    E.g., an IF clause means:

    test the *expression;* if it evaluates to true, execute the *statements.*

Context-free and Context-sensitive Syntax

- The context-free syntax part specifies legal sequences of symbols, independent of their type and scope.
- The context-sensitive syntax part defines restrictions imposed by type and scope.
  - Also called the "static semantics". E.g., all identifiers must be declared, operands must be type compatible, correct #parameters.
  - Can be specified informally or through *attribute*

Compiler and Language Design

- **There is a strong mutual influence:**
  - hard to compile languages are hard to read
  - easy to compile language lead to quality compilers, better code, smaller compiler, more reliable, cheaper, wider use, better diagnostics.
- **Example. Dynamic typing seems convenient because type declaration is not needed. However, such languages are**
  - hard to read because the type of an identifier is not known
  - hard to compile because the compiler cannot make assumptions about the identifier's type

Compiler and Architecture Design

- Complex instructions were available when programming at assembly level.
- RISC architecture became popular with the advent of high-level languages.
- Today, the development of new instruction set architectures (ISA) is heavily influenced by available compiler technology.