Terminology & Basic Concepts

Language Processors

- The basic model of a language processor is the **black box** **translator** (or **transducer**)
- Has one **input stream**, one **output stream**, and a black box (program) that translates between the two

```
input stream --> Black Box --> output stream
```
Language Processors

Kinds of Black Box Translators

• Although all are similar in many ways, we call language processors different things depending on their input and output streams

• A **translator** takes as input a program written in one programming language (the **source language**) and outputs another version of the program written in a different language (the **object** or **target language**)

• A **compiler** is a translator that takes as input a **high level** programming language (such as C, Java or Pascal) and outputs machine or assembly language for a target computer

• An **assembler** is a translator that takes as input **assembly language** and outputs **machine language** for a target computer

• A **transliterator** or **preprocessor** is a translator that translates one **high level** language to another **high level** language
Language Processors

Intermediate Languages

- Often compilers, assemblers and other language processors are implemented in multiple stages, using a separate translator for each stage

- In this case there may be an intermediate form of the program between stages

- An intermediate language is a language used internally in a compiler to represent the program between stages

- An intermediate code is an intermediate language that is the machine language of an imaginary ideal (“virtual”) machine (e.g., Pascal P-code, Java JVM byte code)

- An interpreter is a program that directly executes intermediate code by simulating the hardware of the virtual machine
What is a Compiler?

- We will concentrate in this course on compilers, since all stages except the last are very similar for all high level language processors (interpreters, transliterators, HTML processors, etc.)
- A compiler is a program that translates a high level language (Pascal, C++, Turing, Java, C# etc.) to the machine or assembly language of a target computer.
History of Compilers

First Computers (early 1940’s)

• Programmed by hand in machine language using binary codes input using switches

Assemblers (late 1940’s)

• Replaced binary codes with mnemonic operation names (e.g., “ADD”) and binary memory addresses with location (variable) names (e.g., “A”, “B”, etc.)

ADD A,R2 \rightarrow 011110
010010
History of Compilers

Early High Level Languages (1950’s)

• First high level language compilers (TRANSCODE, FORTRAN) simply handled translation of simple algebraic statements to machine or assembly code

  \[ X = A \times B + C \quad \rightarrow \quad \text{MOV} \quad A, R2 \]
  \[ \quad \text{MUL} \quad B, R2 \]
  \[ \quad \text{MOV} \quad C, R3 \]
  \[ \quad \text{ADD} \quad R2, R3 \]
  \[ \quad \text{MOV} \quad R3, X \]

• Compilers were among the most complex programs of the time - large, challenging and difficult to build

• Since then, advances in both theory (formal languages and automata) and practice (modularity and table-driven methods) have combined to make compilers among the best understood and most elegantly engineered software systems
Goals for a Compiler

The Compiler Itself

• Compilers should be small (use reasonable amounts of memory) and fast (use reasonable amounts of CPU and real time)
  • Classical time / space tradeoff - can make compiler smaller in memory by storing data structures on disk, but that will make it much slower due to increased disk access
• Compilers must be reliable - should handle every input program, every time
• Compilers must be diagnostic - should give informative error messages about exactly what is wrong and where
• Compilers should have a good human interface - it should be easy and obvious to use and control, whether on the command line or in an IDE
Goals for a Compiler

**Generated Code**

- Generated code should also be *small* and *fast*
  - But again, we are faced with a *tradeoff* - for example, *unrolled loops* are faster, but bigger
- There is also a tradeoff between the speed of the *compiler* and the speed of the *generated code*
  - If the *compiler* spends a lot more time on *optimization*, it can make much faster *generated code*
- Generated code should be *reliable* - it should always *work*
- Generated code should be *secure* - it should not allow accidental violation of language constraints (e.g., subscripts out of bounds)
- Generated code should be *diagnostic* - run time failures should be caught and reported in terms of the *original source program*
- Generated code should be *faithful* - the code should do what the user wrote, not something else
Goals for a Compiler

Implementation and Maintenance

• Like any software system, a compiler should be implemented in reasonable time and at reasonable cost

• It is particularly important that compilers be easy to maintain - compilers are critical components in computer use and must be fixed quickly

• Compilers be easy to modify - languages evolve and target machines change over time
Goals for a Compiler

Error Handling

• A compiler should reliably both detect and diagnose language violations in the programs it compiles - it must check every language rule, and should give good advice on what exactly the problem is in each case

• A compiler must recover from detection of errors - that is, it should continue processing the rest of the program to look for other errors if at all possible

• Compilers should correct (but not ignore) simple errors when possible - for example, but supplying missing semicolons when flagging the error
Summary

Language Processors

• Basic model input-translator-output
• Assemblers, compilers, transliterators
• Phased translation, intermediate languages, interpreters
• Goals for a compiler

References

• Text, chapter 1

Next

• Basic structure of modern compilers and interpreters, and the role of their phases