Today’s objectives

- Defining compilers and why we study them
- Defining the high-level structure of compilers
- Associating specific tasks, theories, and technologies with achieving the different structural elements of a compiler
  - And building some initial intuition about why these are needed

What is a compiler?

- A software tool that translates
  - a program in source code form to
  - an equivalent program in an executable (target) form
- Converts from a form good for people to a form good for computers

Examples

- Source languages
  - Java
  - C
  - ML
  - COBOL
  - ...
- Target architectures
  - MIPS
  - x86
  - SPARC
  - Alpha
  - ...

Why study compilers?

- In groups (of 3-5 people), list as many reasons as you can in one minute

  - I’m going to try to do a significant amount of active learning in this course
  - We’ll all need to practice, but the benefits should be real

CSE401’s project-oriented approach

- Start with a compiler for PL/0, written in C++
- We define additional language features
  - Such as comments, arrays, call-by-reference parameters, result-returning procedures, for loops, etc.
- You modify the compiler to translate the extended PL/0 language
  - Project completed in well-defined stages
More on the project

- Strongly recommended that you work in two-person teams for the quarter
- Grading based on
  - correctness
  - clarity of design and implementation
  - quality of testing
- Provides experience with object-oriented design and with C++
- Provides experience with working on a team

Break into groups

- I will present a small program to you, character by character
- In 5 minutes, each group will identify problems that you can see that you will encounter in compiling this program
- Here’s an example problem
  - When we see a character ‘1’ followed by a character ‘7’, we have to convert it to the integer 17.

Structure of compilation

- A common compiler structure has been defined
  - Years and years of deep, difficult research intermixed with building of thousands of compilers
- Actual compilers often differ from this prototypical model
  - The primary differences are the ordering and the clarity with which the pieces are actually separated
  - But the model is still extremely useful
- You will see the structure — to a large degree — in the PL/0 compiler

Front- and back-end

- These parts of the compiler structure are often split into two categories
- The front-end
  - Focuses on (repeated) analysis
  - Determines what the program is
- The back-end
  - Focuses on synthesis
  - Produces the target program that is equivalent to the source program
An example compilation

```markdown
module main;
  var x:int, result: int;
  procedure square(n:int);
  begin
    result := n*n;
  end square;
  begin
    x := input;
    while x <> 0 do
      square(x);
      output := result;
      x := input;
  end;
end main.
```

- A real PL/0 program
- We’ll step through
  - Lexical analysis
  - Syntactic analysis
  - Semantic analysis
  - Storage layout
  - Code generation

Lexical analysis
(AKA scanning and tokenizing)

- Read in characters and
  - Also strip out white space and comments
  - Use regular expressions to
  - Remember the connection
  - Use finite state machines to
  - Between regular expressions
  - With finite state machines

Syntactic analysis
(AKA parsing)

- Turn token stream into tree
  - Syntax using context
  - Define syntax using context
  - EBNF is a common notation
  - Care about abstract syntax
  - Care about statement
  - Care about statement

Semantic analysis
(Name resolution and type checking)

- Given AST
  - figure out what declaration each name refers to
  - perform static consistency checks
- Key data structure: symbol table
  - maps names to information about name derived from declaration
- Semantic analysis steps
  - Process each scope, top down
  - Process body of each scope in context of symbol table

Storage layout

- Given symbol table, determine how and where variables will be
  - What representation is used for each kind of data?
  - How much space does each variable require?
  - In what kind of memory should it be placed?
    - static, global memory
    - stack memory
    - heap memory
  - Where in memory should it be placed?
    - e.g., what stack offset?

Code generation

- Given annotated AST and symbol table, produce target code
- Often done as three steps
  - Produce machine-independent low-level representation of the program (intermediate representation or IR)
  - Perform machine-independent optimizations (optional)
  - Translate IR into machine-specific target instructions
    - Instruction selection
    - Register allocation
Compilers vs. interpreters

- Compilers implement languages by translation
- Interpreters implement languages directly
- Note: the line is not always crystal-clear
- Compilers and interpreters have tradeoffs
  - Execution speed of program
  - Start-up overhead, turn-around time
  - Ease of implementation
  - Programming environment facilities
  - Conceptual clarity

Engineering issues in compiling

- Portability
  - Ideal is multiple front-ends and multiple back-ends with a shared intermediate language
- Sequencing phases of compilation
  - Stream-based vs. syntax-directed
- Multiple, separate passes vs. fewer, integrated passes
- How to avoid compiler bugs?

Objectives: next lecture

- Define overall theory and practical structure of lexical analysis
- Briefly recap regular expressions, finite state machines, and their relationship
  - Even briefer recap of the language hierarchy
- Show how to define tokens with regular expressions
- Show how to leverage this style of token definition in implementing a lexer