Today’s objectives

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

Administrative Details

- Course Web:
  

- Grading
  
  - Homeworks: ~20%
  - Project: ~40%
  - Midterm: ~15%
  - Final: ~25%


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
  - C++
  - LISP
  - ML
  - COBOL
  - ZPL
  - ...

- Target architectures
  
  - MIPS
  - x86
  - SPARC
  - Alpha
  - ...
  - C

Why study compilers?
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 in a team

What's hard about compiling

- I will present a small program to you, character by character
- 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.

Example

```
i 12 1
n 12 7 26 *
t 13 23 i
i 14 1 24 +
i 15 p 25 2
j 16 r 26 )
i 17 i 27 i
i 18 n
p 1 28 t
10 = 29 {
```

• i is the space character
• This is not a PL/0 program!

Structure of compilers

- 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 prototype
  - Primary differences are the ordering and 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

Prototype compiler structure

```
Source Program  
Stream of characters

Lexical analysis

Sequence of tokens

Syntactic analysis

Abstract Syntax Tree (AST)

Semantic analysis

AST + and symbol table

Storage layout

Intermediate code

Optimization

Intermediate code generation

Target Program

Code generation

Executable code

Intermediary representation

Intermediary representation

Intermediary representation

AST + and symbol table

```

These parts are often lumped into two categories
- The front-end
  - Focuses on (repeated) analysis
  - Determines what the program is
- The back-end
  - Focuses on synthesis
  - Produces target program equivalent to source program

**Lexical analysis** (aka scanning and tokenizing)
- Read in characters and clump them into tokens
- Also strip out white space and comments
- Specify tokens with regular expressions
- Use finite state machines to scan
- Remember the connection between regular expressions and finite state machines

**Syntactic analysis example**

**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 declarations in each scope into symbol table for scope
  - Process body of each scope in context of symbol table

**Syntactic analysis** (aka parsing)
- Turn token stream into tree based on the program’s syntactic structure
- Define syntax using context free grammar (CFG)
- EBNF is a common notation for defining concrete syntax
  - Cares about semi-colons, parens, and such
- Parser usually constructs AST representing abstract syntax
  - Cares about statement structures, precedence and such

**An example compilation**
- A real PL/I program
- We’ll step through
  - Lexical analysis
  - Syntactic analysis
  - Semantic analysis
  - Storage layout
  - Code generation

**Syntactic analysis example**

**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 declarations in each scope into symbol table for scope
  - Process body of each scope in context of symbol table

**Syntax analysis example**
Semantic analysis example

```c
int x;
int y(void);
int main(void) {
    double x,y;
    x = x + 5;
    printf("x is %d",x);
    x = y();
    return 1/2 ;
}
```

- Which var with which decl?
- What type?
- Operators legal on those types?
- Coercion?
- Function arg & return types too?
- Overloading?
- goto/case labels unique?

Storage layout

- Given symbol table, determine how and where variables will be stored at runtime
- 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?

Storage layout example

```c
int x;
int y(void);
int main(void) {
    double x,y;
    x = x + 5;
    printf("x is %d",x);
    x = y();
    return 1/2 ;
}
```

- Outer x: 4 bytes, static
- Inner x, y: 8 bytes each on stack
- What address?
- How does printf find its parameters?
- How does main return a value?

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

Codegen example

```c
x = x + y;
```

<table>
<thead>
<tr>
<th>x</th>
<th>142 x</th>
<th>lw $2, 48($p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>143 y</td>
<td></td>
<td>lw $3, 52($p)</td>
</tr>
<tr>
<td>144</td>
<td></td>
<td>add $2, $2, $3</td>
</tr>
<tr>
<td>144 142 + 143</td>
<td>sw $2, 48($p)</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>x = x * 2;</td>
<td>145 x</td>
<td>lw $2, 48($p)</td>
</tr>
<tr>
<td>146 2</td>
<td></td>
<td>li $3, 2</td>
</tr>
<tr>
<td>147 145 * 146</td>
<td>sw $2, 48($p)</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>x += y;</td>
<td>148 x</td>
<td>lw $2, 48($p)</td>
</tr>
<tr>
<td>149 y</td>
<td></td>
<td>lw $3, 52($p)</td>
</tr>
<tr>
<td>150 148 + 149</td>
<td>add $2, $2, $3</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>150</td>
<td>sw $2, 48($p)</td>
</tr>
</tbody>
</table>

Does this structure work well?

- FORTRAN I Compiler (circa 1954-56)
  - 18 person years
- PL/0 Compiler
  - By the end of the quarter, you'll have a working compiler that's way better than FORTRAN I in most respects
    - key exception: optimization
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

Compiler engineering issues
- 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