Course Information

• **Instructor:** Pat Morin  
  morin@scs.carleton.ca  
  – Just "Pat"

• **Office Hours:** Tuesdays 9:00-10:00, 13:30-14:30

• **Webpage:**  
  – Contains all information related to the course

• **Textbook (not compulsory):**  
Course Information

• Grading Scheme:
  – Assignments 4 * 20% = 80%
  – End-of-term exam: 20%

• Grading:
  – Assignments are graded by how well they work, not how much work you put into them
  – A buggy compiler is worse than a missing feature

• Collaboration:
  – Students may discuss assignments, but when it comes time to write code they should do so on their own. No student should ever show another student their code.
Course Information

• Languages and Tools:
  – Programming in Java
  – SSCC parser generator
  – Jasmin JVM assembler

• Environment
  – Examples will be compiled under Linux on the command line
  – I/O will be through System.in/System.out
  – Assignments will generate command line tools
**Definition of a Compiler**

- **What is a compiler?**
  - **Input:** text in language A
  - **Output:** text in language B

- **In this course, A is a programming language and B is a computer (machine) language**
  - Programming languages: Java, C, C++, C#, Eiffel, Lisp, Pascal, Haskell, ...
  - Machine languages: i386, x86_64, PPC, JVM, ...
Structure of a Compiler

• Compilation usually works in (at least) two steps
  – Syntax analysis (tokenization and parsing)
  – Code generation and optimization

• Between the two is an intermediate representation
  – Sometimes called a parse tree or pseudo instructions
Syntax Analysis

- Syntax analysis has two parts
  - tokenization and parsing

![Diagram of syntax analysis process]

- Program text is tokenized by the tokenizer, creating a token stream.
- The token stream is then parsed by the parser to produce an intermediate representation.
- The intermediate representation is then processed by the code generator to produce machine code.
Tokenization

• Converts a character stream into a token stream

```c
int main(void) {
    for (int i = 0;
         i < 10;
         i++) { ...
    }
```
**Parsing**

- Converts a token stream into an intermediate representation
  - Captures the meaning (instead of text) of the program
Compiler Structure

• This structure allows us to reuse compiler components
  – By writing $n$ syntax analyzers and $m$ code generators we get a $nm$ compilers

Diagram:
- C parser
- Java parser
- Eiffel parser
- Pascal parser
- i386 code gen.
- PPC code gen.
- ARM code gen.

Intermediate representation (parse tree)
Code Generation

- Code generation and optimization is the really hard part (to do well)
Code Generation

- Code generation can be done in several phases
  - *Machine independent optimizations* optimize code, without making use of machine-dependent details
  - *Basic code generation* makes no attempt to optimize code
  - *Machine dependent optimizations* optimize code for a specific machine architecture
  - Can be several iterations of each kind of optimization
A Brief History of Compiler Construction

• 1945-1960: Code generation
• 1960-1975: Parsing
• 1975-Present
  – Code optimization
  – Programming language design
  – New programming paradigms
Why Study Compilers?

• A great success story from theoretical computer science
• You may have to write a compiler or interpreter some day
• Parsers appear in a lot of applications
• Translators (file converters) are often required
• Code optimization is still a challenging and active field of research
Why Study Compilers?

- Understanding compiler optimization can improve a programmer's code writing skills
- A programmer will eventually run into a compiler bug, limitation, or "quirk"
  - understanding compilers will help understand what is wrong
- And many more reasons...

http://xkcd.com/371/