Overview of Compilers

- At the most abstract level, a ***compiler*** is simply a tool that translates code written in one programming language (the ***source language***) into code in another language (the ***target language***).

- Most commonly, compilers are understood as tools that translate code written in a human-oriented programming language into the instruction set for a specific machine.

- For example, you've likely used the gcc compiler, which translates code written in C or C++ into instructions for a specific processor, e.g. an x86 processor.

- Translation of this sort is necessary because, as you likely understand at this point in your computer science career, the languages in which we typically write programs are designed to allow humans to easily express sequences of computations, but these languages are not directly executable by any processor.

- Instead, processors typically implement instructions at level of abstraction so low that it is difficult for humans to use these instructions directly. Indeed, a single statement in a human-oriented language might translate into hundreds of machine instructions.

- Compilers bridge this gap, allowing humans to program in a language that's easier to work with and then translate their program into machine instructions to be executed.

- Importantly, though, the target language for a compiler does not have to be a processor's instruction set. Compilers are often used to translate one human-oriented language into another human-oriented language. Such compilers are often called ***source-to-source compilers*** or ***transpilers***.
    - For example a transpiler can be used to convert a program written in Python 2 into Python 3 or a program written in a more modern version of JavaScript (e.g. ES 2015 or ES 2017) into one written in a version of JS that's more widely supported by available web browsers.

- Note, too, that it is important to distinguish between an ***interpreter*** and a compiler. Whereas a compiler takes as input a program written in some language and produces as output the same program in another language, an interpreter takes as input a program written in some language and produces as output the *results* of executing that program.
    - In other words an interpreter *runs* a program.

- Languages like C, C++, and FORTRAN are compiled languages, i.e. languages that are compiled into machine instructions, which are in turn executed.

- Languages like Python and JavaScript, on the other hand, are interpreted.

- Java is a hybrid language: Java source code is compiled into a representation called ***bytecode***, which is then executed by the Java Virtual Machine (JVM), an interpreter for bytecode.
    - In fact, some implementations of the JVM use a scheme called ***just-in-time*** (JIT) compilation, in which some bytecode sequences are compiled at runtime into machine instructions.

Why study compilers?

- You might be asking why we even still study compilers, since good compilers already exist for languages like C, C++, Java, FORTRAN, etc.

- There are a few reasons the study of compilers is still highly relevant today. First, a compiler is a large, complex program, and the design and implementation of a compiler is a great exercise in software engineering.

- Second, compilers encapsulate concepts from across computer science into a single program, including data structures, greedy algorithms and heuristic searches, approximation algorithms, graph algorithms, finite automata, etc. Indeed compilers are a great example of the successful application of theory to practical problems, and you'll see concepts from your CS theory courses applied in this course.

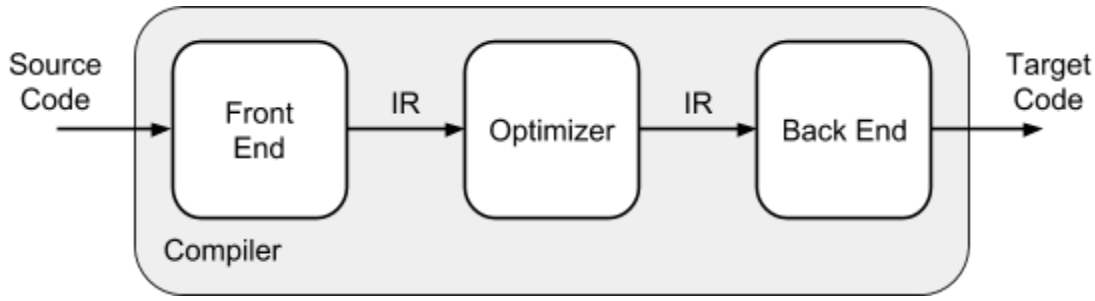
- For the above reasons, this course in compilers serves as a kind of capstone, tying together concepts from across the CS curriculum into a single subject.

- Finally, and perhaps most importantly, new languages are being developed all the time, sometimes for widespread use and sometimes for specific, limited domains, and programs written in these languages need to be executed on real machines. Thus, these languages need either a compiler or an interpreter (the latter of which will necessarily employ many of the same approaches used in the development of a compiler), and these compilers and interpreters need to be written by real people.

- In other words, the practice of compiler design and implementation is far from irrelevant. It is very relevant.

- Indeed, the Rust programming language is a modern success story for compiler development.
    - Mozilla designed Rust and the Rust compiler with the main purpose of completely re-implementing Firefox to run quickly and use less memory on modern machines. The new Firefox, written in large part using Rust, was recently released, and it fulfills many of the goals originally set out when development of Rust began.
    - Mozilla is now positioning Rust as a systems language for modern computers, and many see Rust as eventually supplanting C and C++.
    - Of course, this would not be possible without the highly effective Rust compiler.

What does a compiler look like?



- A modern compiler consists of three primary components: the ***front end***, the ***back end***, and the ***optimizer***.

- The glue that binds all of these components together is the ***intermediate representation*** (or ***IR***), which is the compiler's internal representation of the code being compiled.

- The front end of the compiler is responsible for reading and understanding the source code being compiled. It does things like:
    - Scanning the source code character by character and separating it into individual words.
    - Parsing those individual words into syntactic statements the source language and detecting syntax errors in the source program.
    - Checking type correctness across the source program (depending on the source language's type system).
    - Generating an intermediate representation of the source program to be passed to the optimizer and back end of the compiler.

- The back end of the compiler deals primarily with the target language. It does things like:
    - Select instructions in the target language to implement instructions from the intermediate representation generated by the front end/optimizer.
    - Allocates registers from the target machine to hold values in the generated code, if the target language is a particular processor's assembly language.
    - Schedules instructions in the target language by reordering them to minimize the number of clock cycles wasted waiting for memory accesses, etc., especially if the target language is a processor's assembly language.

- Many (though not all) compilers include an optimizer, which can make one or more passes over the intermediate representation generated by the compiler's front end to try to improve it before it is translated into target-language code by the compiler's back end. The optimizer typically tries to improve the efficiency of the program in one or more ways. Examples of efficiency improvements the optimizer can try to achieve are:
    - Reducing the running time of a program.
    - Reducing the size of the compiled code.
    - Reducing the number of page faults that occur during a program's execution.
    - Reducing the amount of energy used by a machine to run a program.

- Importantly, this decomposition of the compiler into front end, back end, and optimizer can make it easier to ***retarget*** the compiler, i.e. to change either the source or target language.
    - For example, the same front end could be hooked to various back ends to compile code in a single source language into the instruction sets for various different processors.

    - Similarly, different front ends that output the same form of IR could be hooked to a common back end to compile different source languages into assembly code for the same processor.

- Let's look in a little more detail at what each of the compiler's individual components does.

The front end

- The job of a compiler's front end is to read and understand the source code and to generate an IR representation of it, checking for errors in syntax and program construction (e.g. type errors) along the way.

- This job is carried out in two main phases: ***scanning*** (or ***lexing***) and ***parsing***.

- Formally, every source language is a a set of strings (usually an infinite set) defined by a finite set of rules, called a ***grammar***, that describe how a valid program in the source language is constructed.

- A language's grammar might contain a rule like the following:

  AssignmentStatement → `identifier =` *Expression* `;`

- Conceptually, this rule indicates that an assignment statement in the source language can be formed from an identifier token, followed by an equal sign token, followed by an expression in the source language, followed by a semicolon token.
    - The language's grammar may contain other rules describing different forms for an assignment statement, as well.

- This rule contains three types of entities:
    - The → symbol means "derives" and separates the left hand side of the rule (the language construct described by the rule) from its right hand side, which describes how language constructs can be combined to form an instance of the construct on the left hand side.
    - The tokens `identifier`, `=`, and `;` are specific syntactic categories. These are like parts of speech in a natural language, describing the specific role a given word in the language plays.

        - In this example, the `identifier` syntactic category is one that could potentially be applied to many different words in the language, while there may be only one specific word to which each of the `=` and `;` categories can be applied.
    - *AssignmentStatement* and *Expression* are syntactic variables. The *AssignmentStatement* variable serves as the rule's left hand side, and the *Expression* variable serves as a placeholder for further syntactic categories.
        - The language's grammar must also contain rules describing how an *Expression* can be derived.

- The first phase of the compiler's front end is the scanner. The role of the scanner is to identify distinct words in the source code and to classify each word with a syntactic category.

- Specifically, the scanner takes as input a stream of characters (the source code) and outputs a stream of classified words, each of which is a pair (*c, s*), where *c* is a word's syntactic category and *s* is the word's spelling.

- For example, say the scanner reads a source code sentence like this:

```
a = b + 1;
```

- Then the scanner would output something like the following list of words: (`identifier, "a"`), (`=, "="`), (`identifier, "b"`), (`+, "+"`), (`number, "1"`), (`;, ";"`).

- The second phase of the compiler's front end is the parser. The role of the parser is to match the categorized words output by the scanner against the set of grammar rules specifying the source language's syntax. The parser uses the information encoded in the matched rules to generate an intermediate representation to be passed on to the compiler's optimizer and back end.

- For example, the source language's grammar might include the following rules (among many others):

    1 *Statement* → *AssignmentStatement*
    2 *AssignmentStatement* → `identifier =` *Expression* `;`
    3 *Expression* → *Expression Op Term*

4 *Expression* → *Term*
5 *Op* → +
6 *Op* → -
7 *Term* → `identifier`
8 *Term* → `number`
...

- Given these rules, the parser, starting with the syntactic variable *Statement*, might arrive at the following derivation for our example sentence, given the above scanner output:

Rule	Sentence Prototype
–	*Statement*
1	*AssignmentStatement*
2	`identifier =` *Expression* `;`
3	`identifier =` *Expression Op Term* `;`
4	`identifier =` *Term Op Term* `;`
7	`identifier = identifier` *Op Term* `;`
5	`identifier = identifier +` *Term* `;`
8	`identifier = identifier + number ;`

- In other words, starting with the syntactic variable *Statement* and applying grammar rules in the order {1, 2, 3, 4, 7, 5, 8}, the parser is able to generate a prototype sentence that matches the sequence of categorized words output by the parser.
    - In the course of constructing this derivation, the parser will also be constructing an IR representing the sentence being parsed. There are many different kinds of IR that could be used here.

- In this way, the parser confirms that the sentence "`a = b + 1;`" does belong to the set of strings comprising our example language. However, this does not mean that the sentence is a valid one.

- For example, if the identifier `b` refers to an object of some class, the sentence could be invalid, even though it is syntactically well-formed, since the language might not allow adding an object and a number.

- Depending on the source language's type system, the compiler's front end might also be responsible for checking type consistency, to catch errors like this in the source code. Checks like this can be done directly within the parser, or they can be done in a separate pass.

- Importantly, there are different approaches to parsing, and the type of parser being used places constraints on the language that can be recognized and how the grammar expressing the syntax rules for that language must be formulated.

The optimizer

- A compiler's front end typically considers statements in the source code one at a time, in order, without worrying about the surrounding context of each statement.

- The job of the optimizer is to analyze the IR generated by the compiler's front end to discover facts about the context in which the code runs in order to rewrite the code to compute the same thing in a more efficient way.
    - While the classic notion of efficiency here is with regards to a program's runtime, the optimizer can also seek to reduce the size of the compiled code, to reduce the energy consumed executing the compiled code, etc.

- For example, consider the following code snippet:

```
a = 1
b = ...
c = ...
for i = 1 to n
  a = a * 2 * b * c * i
```

- In this snippet, the values of `a` and `i` are the only ones that change within the loop, while `2`, `b`, and `c` are invariant. If the optimizer can discover this fact, it could rewrite the code as follows:

```
a = 1
b = ...
c = ...
t = 2 * b * c
for i = 1 to n
  a = a * t * i
```

- By doing this, the optimizer would reduce the number of multiplications performed by this code from 4`n` to 2`n` + 2.
    - Note though that the code is slightly larger and needs slightly more space to store the value of `t`.

- Within a compiler, optimizations typically consist of an analysis and a transformation. There are many kinds of analysis and many kinds of transformations that can be used.

The back end

- Finally, the compiler's back end traverses the IR output by the front end or optimizer and produces code in the target language. This can include:
    - Selecting target-language operations to implement operations specified in the IR.
    - Orders the selected target-language operations so they will run efficiently.
    - If the target language is an assembly language for a specific machine, allocating actual machine registers to hold values in the selected target-language instructions.

- The first phase of the compiler's backend is the ***instruction selector***, which maps the IR generated by the front end/optimizer to the target language.

- For example, say the front end/optimizer generated the following linear IR for the statement `a = a * 2 * b * c`:

```
t0 = a * 2
t1 = t0 * b
t2 = t1 * c
a = t2
```

- The instruction selectormight select the following instructions to represent this IR (in ILOC, the assembly-like language used in our textbook):

```
loadAI    r_arp, @a ⇒ r_a    // load a
loadI     2         ⇒ r_2    // "load" constant 2
loadAI    r_arp, @b ⇒ r_b    // load b
loadAI    r_arp, @c ⇒ r_c    // load c
mult      r_a, r_2  ⇒ r_a2   // multiply a * 2
```

```
mult      r_a2, r_b ⇒ r_a3    // multiply (a * 2) * b
mult      r_a3, r_c ⇒ r_a4    // multiply (a * 2 * b) * c
storeAI   r_a3      ⇒ r_arp, @a   // write result back to
a
```

- Here, each `r` designates a unique register. `r_arp` is a special register containing an address known as the ***activation record pointer***, which marks the start of the data storage for the current procedure. The values `@a`, `@b`, and `@c` are offsets from this address.

- This sequence of operations is a straightforward mapping from the IR to the target language, but this is not the only sequence of instructions that could have carried out the same computation, e.g. taking advantage of special operations available in the target language or other knowledge about the target machine to select more efficient operations.

- For example, if the target language has a multiplication instruction that directly multiplies a value in a register by a constant value, the `loadI` instruction that loads the value 2 into a register could be dropped, or if the addition instruction is faster than multiplication, the instruction selector could replace `mult r_a, r_2` with `add r_a, r_a` (also avoiding the need to load the value 2).

- Similar opportunities arise during instruction selection when transpiling to a different source language instead of compiling to an assembly language.

- When compiling to an assembly language for a specific processor, the ***register allocator*** is the phase of the compiler's back end that follows the instruction selector.

- In particular, the instruction selector typically doesn't worry about how many registers the target machine has; it simply uses an infinite supply of ***virtual registers***. The register allocator's job is to map these virtual registers to real physical registers from the target machine's limited set.

- To do this, the register allocator must decide at each point in the code what values will live in registers and what values will be kept in memory.

- This process involves many tradeoffs and represents a challenging optimization problem.

- The final phase of the compiler's back end is the ***instruction scheduler***, which, again, is typically only used when the target language is a specific machine's assembly language.

- Importantly, the order in which instructions are executed can have a significant impact on how long a program takes to execute.

- For example, many processors can initiate new operations while a long-running operation, such as a memory load, executes, provided that those new operations don't rely on the result of the long-running operation.

- To take advantage of this property, the instruction scheduler can reorder the operations to minimize the amount of time wasted waiting for long-running operations to complete while still ensuring that the computation produces the same result.

- Again, instruction scheduling is a challenging optimization problem and presents various tradeoffs, particularly with the register allocator, since reordering operations can sometimes require using more registers.