CS101
Introduction to Programming Languages and Compilers

In this handout we'll examine different types of programming languages and take a brief look at compilers. We'll only hit the major highlights here – the textbook has additional details in chapters 8 and 9.

Programming Languages

So far, we have examined machine code, taken a brief look at assembly language, and have taken a more detailed look at Java. It so happens that there are numerous programming languages in addition to Java. The major categories are:

- Procedural
- Functional
- Logic
- Special-purpose (e.g. web-based, database, etc.)
- Parallel

Let's briefly look at each of these categories and see some properties of programming languages in each one. If you continue to take the programming languages course at UAA, then you will get the opportunity to write programs in all of these areas.

Procedural Languages

A procedural language is one in which the program consists of statements that are executed sequentially. This corresponds fairly directly to the actual machine code and the way the von Neumann machine operates. For example, if we have the following statements in Java:

```
x = x +1;
y = y * x;
if (a < b) {
        System.out.println("Foo");
}
else {
        System.out.println("Bar");
}
```

The sequential model executes the statement "x=x+1" first. When that is finished, the next statement "y = y * x" is executed. When that is finished, the if statement is executed, and so on.

Although there are many different types of procedural languages, they all share the sequential execution model in common. There may also be differences in some features; for example object-oriented vs. no objects.

Often the main differences are in syntax only, making it fairly easy to learn other programming languages once you have learned one.

Here are some common procedural languages:

FORTRAN - This language is an acronym for FORmula TRANslation. It was quite popular in the 50's and 60's but today is rather outdated and is mostly used with supercomputing and engineering applications.

COBOL - This language is an acronym for COmmon Business Oriented Language. As the name implies, it was designed for business applications such as storing databases or formatting reports. COBOL was also popular in the 60's but is rather outdated by today's programming language standards.

Pascal - This language was invented by Niklaus Wirth as a teaching language and is named after mathematician Blaise Pascal. It was popular in the 80's and early 90's but is no longer used very often for teaching since C and C++ were more common in industry. Here is a sample of Pascal code to sum the numbers from 1 to 10:

```
var
      x, sum:integer;
begin
      x:=1;
      sum:=0;
      while (x <= 10) do
      begin
            sum:=sum+x;
            x:=x+1;
      end;
end
```

Here is the equivalent code in Java:

```
int x,sum;

x=1;
sum=0;
while (x <= 10) {
      sum = sum + x;
      x = x + 1;
}
```

As you can see, the logic of the two programming languages is quite similar. There are some differences such as the words "begin" and "end" in place of "{" and "}" and also a different way of declaring variables. However, these differences are typically quite easy to learn once you have learned one programming language as a reference point!

C/C++ - These languages were developed at Bell Laboratories as a way to develop the Unix operating system. They are still quite popular today and are used quite frequently to develop standalone applications that run quickly, as this language can be

combined with assembly code and presents many opportunities for optimization. Most applications that must run quickly are written in C or C++. For example, the Linux operating system and the majority of Microsoft Windows is written in C/C++. The C language was developed first. C++ was developed later and extends C to include object-oriented behavior. A large portion of Java was based on the syntax used in C++. In fact, the snipped of code shown above for summing the numbers from 1 to 10 will also compile and run in C or C++!

Functional Languages

A functional language performs all tasks as functions. This can make the syntax very easy to learn and has some nice mathematical properties as we can extend the lambda calculus to our computer programs. LISP and Scheme are two functional languages that are used today.

Here is the syntax of a Lisp program:

```
(function-name  function-argument-1  function-argument-2 …function-argument-N)
```

That is the whole syntax of everything in Lisp! The harder part is learning what functions and arguments are available. For example, say that "plus" is the function to add two numbers and "multiply" is the function to multiply two numbers.

The following code would compute the sum of 4 and 5:

```
(plus 4 5)
```

This code would compute the product of 2 and 3:

```
(multiply 2 3)
```

We can imbed functions as arguments of another function. The following computes the value of (4+5) * 3:

```
(multiply (plus 4 5) 3)
```

The following computes the value of ((4+5)*3) + 1:

```
(plus (multiply (plus 4 5) 3) 1)
```

Functional languages are popular for certain verification and artificial intelligence tasks.

Logic Languages

Logic programming is quite different from procedural programs in that programs are based on predicate calculus. Rules are given to the system together with asserted facts, and the system will infer or deduce other facts. These have been used to create "expert systems" where a human inputs knowledge about a particular domain, say, information about what symptoms may be predictive of various cancers, and then users will give their symptoms and have the system infer whether or not they have cancer using the same logic that the doctor might use.

Here is a very simple set of knowledge we might write in a logic language:

- Baneberry is poisonous.
- Baneberry is a plant.
- A poisonous plant should not be eaten.

In this example we have given the system two facts: that baneberry is poisonous and that baneberry is a plant. Using the rule that a poisonous plant should not be eaten, the logic language is able to infer that baneberry should not be eaten.

Prolog is the most popular logic programming language. It has been used for expert systems, formal verification, theorem-proving, constraint satisfaction, and many artificial intelligence applications.

Special-Purpose Languages

As the name implies, special-purpose languages are designed with a particular purpose in mind. Here are just a few such languages:

SQL - SQL is the Structured Query Language. It is used to query databases and is often used in conjunction with other programming languages. This is quite common for many business applications. For example, consider a table named "PRODUCTS" that contains a list of products you are selling:

Product ID	**Product Name**	**Price**	**Description**
1031	Paper clips	4.99	100 colored paper clips
4912	Bubble gum	0.50	Chewy and yummy
2019	Pet food	8.99	Specially formulated for smooth coats
9102	Jet engine	450,000	For all your aviation needs

Perhaps you would like to search the database for only those products that cost less than 5 dollars. In SQL one can write a query to retrieve just those items:

```
SELECT * FROM PRODUCTS WHERE (Price < 5);
```

The "*" retrieves all columns that match the condition of price less than $5. This query returns the following table:

Product ID	Product Name	Price	Description
1031	Paper clips	4.99	100 colored paper clips
4912	Bubble gum	0.50	Chewy and yummy

Next, using a programming language, we could perform operations like scan through the resulting table and perhaps format, add, or retrieve specific pieces of data we are interested in.

PERL - Perl is an acronym for Practical Extraction and Report Language. It is a fairly high-level language that is very efficient for processing text files and matching patterns of text. It is often used for system administration and sometimes used with web page scripts.

HTML - We have already discussed HTML previously. This is not really a programming language, but it is a language that specifies the layout of items on a web page.

PHP - PHP is an acronym for PHP: Hypertext Processor. It is a language designed specifically for the web. It allows the programmer to insert Java-like code directly into web page scripts and is often used in conjunction with HTML and SQL to integrate web pages with databases.

Parallel Languages

Parallel programming languages are typically modifications of existing languages to provide better performance on parallel machines. A parallel machine is one with multiple processors. If a machine has multiple processors, it is able to perform multiple tasks simultaneously for an improvement in speed. However, it can often be a difficult task to break a program up so that it can be executed in parallel.

Consider our program that computes the sum of numbers 1-10:

```
x=1;
sum=0;
while (x <= 10) {
      sum = sum + x;
      x = x + 1;
}
```

The value of variable sum each iteration through the loop is dependent upon the value of x. That is, the future value of sum is dependent on the current value of x. This is called a *dependency* because sum depends on x. When we have dependencies like this, even if we had 10 processors, there is not a good way to utilize all ten processors to compute the answer. We are stuck doing the job with a single processor.

However, instead consider the following problem:

Given 10 values, V_1, V_2, … V_{10}
Add one to each value

In our code, let's represent V_1 as V[1], V_2 as V[2], etc. Here is code to perform this task:

```
x=1;
while (x <= 10) {
      V[x] = V[x] + 1;
      x = x + 1;
}
```

This program will work, but what if we had ten processors? Then we could have processor one compute V[1] = V[1] + 1 at the same time processor two computes V[2] = V[2] + 1, all the way up to processor ten, which computes V[10] = V[10] + 1. In one step we could complete the operation, instead of requiring ten steps!

A parallel programming language lets us tell the computer what processors should be working on what tasks. For example the above code might be expressed equivalently on a parallel processing machine as:

```
Parallel (x = 1 to 10) {
      Processor[x] : V[x] = V[x] + 1;
}
```

Compilers

The previous discussion on programming languages is nice, but ultimately our machines can only run machine code. This means that whatever we write our program in must somehow be converted into machine code. The process of converting from a high-level programming language to machine code is the job of a compiler.

A compiler has a much more difficult job to do than an assembler. Recall that an assembler converts assembly code mnemonics into machine code. This is relatively fast and easy because there is a 1 to 1 relationship between assembly code and machine code:

To compute D = (B + C) * A, here is some hypothetical machine code and assembly code:

Assembly Code	Corresponding Machine Code
LOAD C	1011010110001
ADD B	1110101010010
MUL A	1111101110001
STORE D	0010010111011

The assembler must merely convert each assembly code into corresponding machine code.

This job is much harder for the compiler, because the relationship between high-level code and machine code is 1 to many. That is, one statement of high-level code can result in many machine instructions:

Java Code	Corresponding Machine Code
D = (B+C)*A;	1011010110001
	1110101010010
	1111101110001
	0010010111011

As another example, consider the Java statement:

System.out.println(“Hello world!”);

This single statement requires a large number of machine instructions:

1. Get first character (e.g. “H”)
2. Move this character to the output console to display it
3. Check if we have printed the last letter, if not, get the next character and goto step 2

Once again, the compiler has the task of converting statements in the high-level language like Java into the corresponding machine instructions.

There are four phases to the compilation process:

1. Lexical Analysis
2. Parsing
3. Code Generation
4. Optimization

Lexical Analysis

This is the first phase in compiling a program. It involves scanning through the source code and turning the source code into tokens, where each token is a "word" of interest to the compiler. For example, consider the following snippet of Java code:

```
int num;
num = (10 + x) * 3;
```

Lexical analysis turns this sequence of text into meaningful terms:

Term Number	Token
1	int
2	num
3	;
4	num
5	=
6	(
7	10
8	+
9	x
10	)
11	*
12	3;

To perform lexical processing requires some knowledge about what characters belong together and which do not. For example, we must know that the "1" and "0" in 10 belong together as a single number, but that "3" and ";" are separate tokens.

Parsing

Once the input has been lexically analyzed, the result is passed into the parser. The parser takes the stream of input tokens and determines if the syntax is valid. This is similar to the process of parsing a sentence in English. For example, below is a very simple grammar:

Sentence → Noun Verb-Phrase
Verb-Phrase → Verb Noun
Noun → { Kenrick, cows }
Verb → { loves, eats }

This grammar says that a sentence is formed by starting with a Noun and following it with a Verb-Phrase. For the Noun, we are allowed to select either “Kenrick” or “cows”. For the Verb-Phrase, we must select a Verb followed by a Noun. The Verb can be either “loves” or “eats”, and once again the Noun must be either “Kenrick” or “cows”.

Using this simple grammar our language allows the construction of the following sentences:

Kenrick loves Kenrick
Kenrick loves cows
Kenrick eats Kenrick
Kenrick eats cows
Cows loves Kenrick
Cows loves cows
Cows eats Kenrick
Cows eats cows

The above sentences are “in” the language defined by the grammar. Sentences that are not in the language would be things like:

Kenrick loves cows and kenrick.
Cows eats love cows.
Kenrick loves chocolate.

The first two sentences are not possible to construct given the grammar. The last sentence uses a word (chocolate) that is not defined as a word in the grammar.

We use the same process with programming languages. Each programming language has its own grammar that determines how to parse the tokens generated by the lexical analysis. This grammar determines what syntactically valid computer programs look like.

Consider part of a grammar for an arithmetic expression. Here are a few sample expressions for addition that we might like to handle:

30
30 + 45
30 + 45 + 55
30 + 12 + 102 + etc.

This expression might be used in an assignment statement, e.g.:

x = 30;
x = 30 + 45;
etc.

We can write a grammar to determine valid arithmetic expressions for addition. For starters, we have:

expression → number

That is, an expression can be a number.
But we can also add numbers together. We can represent this by a grammatical rule to add two expressions:

expression → number OR
expression → expression + expression

To use the grammar, we replace occurrences of "expression" on the right hand side of the rule with an entire rule for "expression".

For example, to parse "30 + 45 + 55" first start with "expression" by itself:

expression

Then use the rule "expression → expression + expression" to replace the single expression to get:

expression + expression

For the first expression, use the rule "expression → number" to replace expression:

number + expression

Number matches up with "30":

30 + expression

For the remaining expression, use the rule "expression → expression + expression":

30 + expression + expression

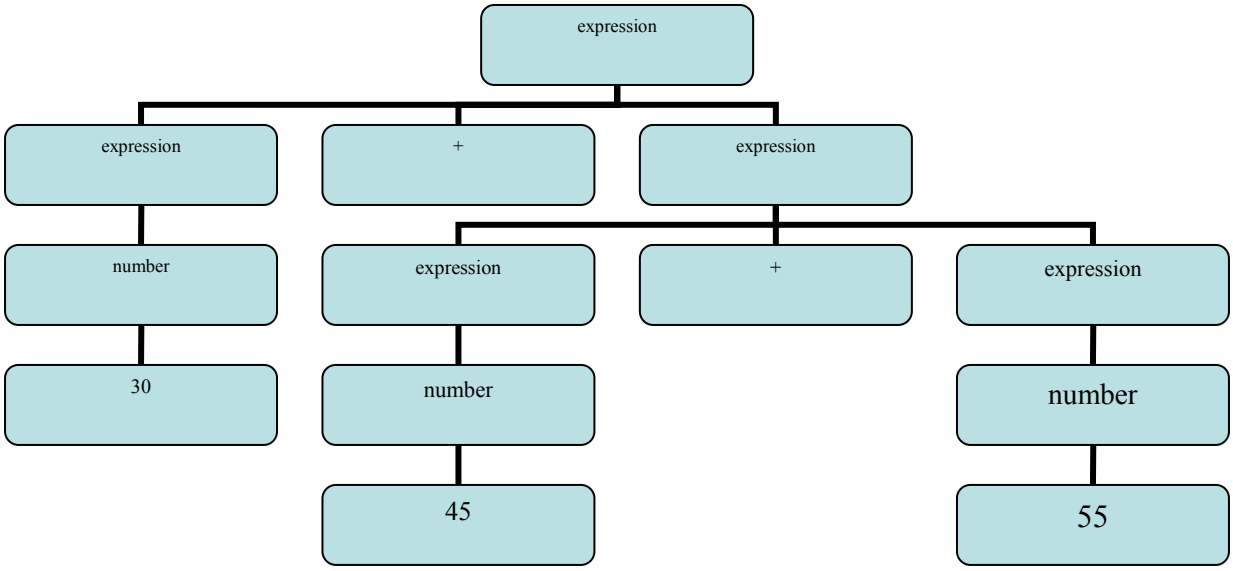
Now, both expressions are replaced by number:

30 + number + number

Finally, each number matches up to 45 and 55:

30 + 45 + 55

We have just parsed the input and determined that we have a complete expression, 30+45+55. We can visualize the parsing process by a parse tree:



This forms an upside-down tree, much like a family history tree. Each lower level of the tree represents the rule we used to replace the node above until we end up with the actual symbols of the program input on the bottom.

Code Generation

After we have parsed our program and determined what "part of speech" (e.g., what is an expression, what is an assignment statement, etc.) each section corresponds to, we can start to generate machine code.

Code generation requires examination of the semantics of the program. For example, consider the English sentence "Colorless green ideas sleep furiously." This sentence is grammatically correct (i.e. the syntax is fine), but it is meaningless semantically. We could have the same thing in a computer program – the syntax may be correct, but the meaning of what the code should do is confusing or meaningless.

Let's say that our parsing process has evaluated some code such as:

```
x = number + value;
```

This has been parsed into:

```
<VARIABLE> = <VARIABLE> + <VARIABLE>
```

The code generation step might decide to place variable *x* in register 1, variable *number* in register 2, and variable *value* in register 3. The compiler can then generate code that completes this evaluation, such as the following assembly language code:

```
ADD R2, R3      // Adds contents of R2 and R3 and puts result in Accumulator
MOV R1, Acc     // Copy value in accumulator to register 1
```

The resulting assembly code corresponds directly to machine code that performs the desired operation. As you might imagine, a very large number of rules and logic is needed to determine the proper machine code for the many possible constructs we can create with most programming languages.

Optimization

The last phase is code optimization. In many cases, a compiler can re-arrange bits of code to make the program run more efficiently. While there are numerous ways that code can be optimized, here are just a couple to give you an idea of what compilers can do:

1. Evaluate constants.

Consider the following code:

```
x = 3 * 4;
```

This could generate machine code that actually multiplies 3 by 4 and puts the result in variable x. The multiplication and the copying into x would be performed while the program runs. However, the value that gets copied into x is always going to be 12. The smart compiler will evaluate 3 * 4 during compilation time, and then during runtime simply copy the value 12 into x. This saves time while running the program that would otherwise be spent multiplying the values together.

2. Replace code with something equivalent yet faster.

Consider the following code:

```
x = x * 2;
```

This could generate machine code that multiplies x by 2 and stores the result back in x. However, if x is an integer, we can get the same effect by shifting each bit of x to the left by one and copying a zero into the right. This is an artifact of the way numbers are represented in binary. For example:

```
1 in binary = 0001
2 in binary = 0010      ← shifted 0001 to the left one bit
4 in binary = 0100      ← shifted 0100 to the left one bit

3 in binary = 0011
6 in binary = 0110      ← shifted 0011 to the left one bit
```

Shifting each bit to the left by one position is much faster than performing a multiplication routine. The smart compiler would replace x = x * 2 with a "Shift x left by 1 bit" instruction instead.

3. Eliminate unnecessary operations

Consider the following loop:

```
x = 1;
while (x < 10000) {
      sum = sum + x;
      z = 3;
      x ++;
}
```

There is nothing wrong with this code, but it is not very efficient. The loop is executing 10,000 times. However, inside the body of the loop we are constantly setting variable *z* to 3. We are doing this 10,000 times! Logically we can get the same effect by simply setting *z* to 3 once after the loop is over:

```
x = 1;
sum = 0;
while (x < 100) {
      sum = sum + x;
      x ++;
}
z = 3;
```

This optimization saves us from executing 10,000 needless assignment statements during execution.

As you can see, there are many ways we can optimize programs. While compilers can do many of these things for us, it is much safer if the programmer can write the code with optimizations in mind. In general, a smart programmer will find many more optimizations than a compiler is able to find.

Here we have only looked at the major tasks that a compiler performs. For more details, see the textbook. There is also an entire computer science course that deals solely with compilers.