Programming Languages | Niall Williams

January 17, 2018 (Class notes)

Why so many languages?

1) **Different tools for different jobs**
    - Internet scraping/interaction (Ruby, javascript, PHP, etc)
    - Scientific computations (R, Fortran, etc)
    - Education (Pascal, etc)
    - AI (LISP, etc)
    - Device control/OS calls (C, etc)
    - COBOL ("**Co**mmon **B**usiness **O**riented **L**anguage")
    - Ada (Developed by the national defense system)
    - String manipulation/Pattern matching (Python, Perl, etc)
2) **Evolution** - There is no perfect language, so new languages may be developed to improve upon others and fill certain voids that exist in the programming language world. Different languages are often influenced by other languages in terms of syntax or design. Nowadays, nobody really tries to create a language that "does everything."
3) **Commercial Interests** - Companies may develop their own languages to make developing their tools easier.
    - C# (Microsoft)
4) **Different ways to think about programming**
    - Object Oriented
    - Recursion vs looping
5) **Special Hardware**

Why study programming languages?

"If a languages doesn't affect the way you think about programming, it's not worth knowing"
-Alan Perlis

1) **Learn new ways to think about programming**
2) **Makes it easier to learn a new language**
3) **Improve your own programming**

Features of successful languages include:

- **Orthogonality** - Everything in the languages works the same way. For example, the output system of Java always uses the `toString()` method to print out anything.
- **Expressiveness** - Think rich operators, libraries. Allows you to do a lot of things easily.
- **Portability** - Think Java. Designed to be portable (work on any hardware).

- **Efficiency** - Think C/C++. Note that sometimes, you must trade off either portability or efficiency to have the other. C/C++ are not very portable (<= and >= operators can yield different outputs depending on the hardware).

Three aspects of PL study

1) **Syntax**
    a) Grammar
    b) Scanning/parsing
    c) Compiling/interpreting
2) **Semantics -** Refers to the meaning of expressions and how the language resolves meanings. Think of passing by reference vs passing by value, name bindings, type systems, scope resolution, etc.
3) **Pragmatics** - Compiled vs interpreted languages, memory management

Dijkstra says the birth of CS was with ALGOL60.
The birth of the theoretical side of CS, however, was *much* earlier. It comes from the world of logic, as logicians were trying to determine what sort of things are computable (deduced, calculated) vs non-computable. Some of these logicians is Alan Turing from the 1930s, Frege from the 1890s, and Church from the 1920s. These logicians and their work essentially created the first foundations of the different families (taxonomies) of programming languages.

-Homework: Read chapter 1-

January 18, 2018 (HW notes)

Chapter 1

1.1 - Reasons for Studying Concepts of Programming Languages

- **Increased Capacity to express ideas** - The language in which we develop software places limits on the kinds of control structures, data structures, and abstractions we can use. Thus, the forms of algorithms we can construct are similarly limited. By studying PL, we get a greater awareness of different PL features which can reduced these limitations in our software development. We can often translate or simulate language constructs from other languages into languages that do not explicitly support these constructs.
- **Improved background for choosing appropriate languages** - By being more aware of different PLs and what aspects make PLs stronger and weaker in various areas, we are better able to choose the appropriate PL for the given task at hand. While it's true that features from one language can often be simulated in another language, sometimes it's best to use the language where this feature is integrated directly into the language, because it's more elegant and safer.

- **Increased ability to learn new languages** - Continuous learning is essential in software development. Once we have a thorough understanding of the fundamental concepts of languages, it becomes much easier to see how these concepts are incorporated into the design of whichever language one is learning. For example, learning Java is much easier once you understand OOP. Additionally, learning the vocabulary and fundamental concepts of PLs allows us to read and understand PL descriptions and evaluations.
- **Better understanding of the significance of implementation** - When we have a good understanding of PL implementation issues, it lets us more easily choose the correct language for a job, as well as lets us design our programs intelligently. We can purposefully avoid code structures that would exacerbate the shortcomings of a langage once we fully understand these shortcomings and how they are implemented.
- **Better use of languages that are already known** - By studying the concepts of PLs, we can learn about previously unknown and unused parts of languages that we already use, which leads to more diverse and potentially more efficient code.
- **Overall advancement of computing** - If every programmer fully understood the strengths and weaknesses of every language, we would use the best language for every given job. This would help advance computing because we would not be slowed down by the weaknesses of another language that may have been chosen for a job if we were not educated on all aspects of PLs.

1.2 - Programming Domains

Computers are used in every discipline. Each discipline requires different things out of a computer. For example, animators require computers that are capable of rendering complex 3D scenes (strong graphics processing), while financial firms need computers that are capable of computing large calculations efficiently and reliably. Because of this, we have developed various PLs with different needs in mind. Some of the most notable applications of PLs are as follows:

Scientific applications

The first language made for scientific applications was Fortran. Scientific applications required large numbers of floating-point arithmetic computations, and commonly used arrays and matrices. Thus, the early PLs designed for scientific applications we designed with these needs in mind.

Business Applications

COBOL is probably still the most commonly used language for business applications. Business languages are characterized by facilities for producing elaborate reports, precise ways of describing and storing decimal numbers and character data, and the ability to specify decimal arithmetic operations.

Artificial Intelligence

AI is a broad area of computer applications and is characterized by the use of symbolic (rather than numeric) computations. By this we refer to the manipulation of symbols. Symbolic computation is more conveniently done with linked lists of data rather than arrays. The first widely used PL for AI applications was the functional language Lisp. Then they moved onto logic programming using Prolog. Recently, some AI applications have been written in languages such as C.

Web Software

Notable languages here are JavaScript and PHP. In web development, dynamic content needs to be computed and displayed on many different types of machines. Note that HTML is not a PL.

1.3 - Language Evaluation Criteria

We must define a set of evaluation criteria when evaluating the different facets of PLs. Note that not all the criteria have equal importance.

Readability

This is one of the most important criteria for judging a PL. We need good readability because of code maintenance. Around the 1970s, we started to focus more on the human experience rather that the machine experience when designing PLs. Readability depends on the context of the problem domain. For example, if a program that describes a computation is written in a language not designed for such use, the program may be unnatural and convoluted, which makes it unusually difficult to read. The following bullets are characteristics that contribute to the readability of a PL:

- **Overall Simplicity** - A language with a large number of basic constructs is more difficult to learn that one with a smaller number. When we have to use a large language, we often learn a subset of the language and ignore its other features. Readability problems occur whenever the program's author has learned a different subset from that subset with which the reader is familiar. A second complicating characteristic of a PL is **feature multiplicity**, which refers to having more than one way to accomplish a particular operation. Thirdly, **operator overloading** can cause problems. This is when a single operator symbol has more than one meaning. This is often useful, but can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly. For example, it is clearly acceptable to overload + to use it for both int and floating-point addition. But suppose the programmer defined + used between single-dimensioned array operands to mean the sum of all elements of both arrays. The usual meaning of vector addition is quite different from this, this unusual meaning could confuse readers.

- **Orthogonality** - Orthogonality in a PL means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Furthermore, every possible combination of primitives is legal and meaningful. For example, consider data types. Suppose a language has 4 primitive data types (int, float, double, and character) and 2 type operators (array and pointer). If the two type operators can be applied to themselves and the four primitive data types, a large number of data structures can be defined. **The meaning of an orthogonal language feature is independent of the context of its appearance in a program.** Orthogonality follows from a symmetry of relationships among primitives. A lack of orthogonality leads to exceptions to the rules of the language. For example, imagine pointers can point to all data types except arrays. This cuts out maybe potentially useful user-defined data structures. **Orthogonality is closely related to simplicity**: **the more orthogonal the design of a language, the fewer exceptions the language rules require**. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand. Simplicity in a language is at least in part the result of a combination of a relatively small number of primitive constructs and a limited use of the concepts of orthogonality. Some believe that functional languages are possibly the best in terms of overall simplicity, because they can accomplish everything with a single construct, the function call, which can be combined simply with other function calls. However, efficient, has prevented function languages from becoming more widely used.
- **Data Types** - Suppose we are forced to use ints to represent an indicator flag because the language being used has no Boolean type. A statement like `timeOut = 1` is unclear, but `timeOut = true` is perfectly clear.
- **Syntax Design** - The syntax of the elements of a language has a significant effect on the readability of programs. **Special words** greatly influence program appearance and thus influence program readability. If all code blocks end in the same special word or character (like a brace), it becomes harder to read the language. The tradeoff here is that this harder-to-read language will have less special, reserved words. **Form and meaning** also helps with readability. Designing statements to that their appearance indicates their purpose helps with readability. **Semantics (meaning) should follow directly from syntax (form).**

<u>Writability</u>

Writability refers to how easily a language can be used to create programs for a chosen problem. There is a lot of overlap in the characteristics that affect readability and writability, since the person writing a program has to often re-read sections of the program that were already written.

We must be sure not to compare the writability of two languages that were intended for different problem domains. Visual BASIC is much better for creating a program that has a GUI, while C is bad at this. The following are the most important characteristics influencing the writability of a language:

- **Simplicity and Orthogonality** - If a language has a large number of different constructs, programmers might not be familiar with all of them. This can lead to a misuse of some features and a disuse of others that may be more elegant or efficient than those being used. It may even result in using unknown features accidentally, leading to unexpected results. Thus, a smaller number of primitive constructs and a consistent set of rules for combining them (ie orthogonality) is much better than having a large number of primitives.
- **Expressivity** - This can refer to several different characteristics. In a language such as APL, it means that there are very powerful operators that allow a great deal of computation to be accomplished with a very small program. More commonly, it means that a language has relatively convenient, rather than cumbersome, ways of specifying computations. For example, `count++` is more convenient than `count = count + 1.` The inclusion of `for` statements makes it easier to write counting loops than the inclusion of `while` loops. All of these increase a language's writability.

Reliability

A program is considered reliable if it performs to its specifications under all conditions. The following all have significant effects on the reliability of programs:

- **Type checking - Type checking refers to testing for type errors in a given program, either by the compiler of during program execution.** Run-time type checking is expensive, so compile-time type checking is more desirable. The earlier errors are detected, the less expensive it is to make the required repairs. A lack of consistent type checking can also make it very difficult to find bugs in code.
- **Exception handling** - The ability of a program to intercept run-time errors, take corrective measures, and then continue is an obvious aid to reliability. This is known as exception handling.
- **Aliasing - Aliasing is having two or more distinct names in a program that can be used to access the same memory cell.** It is now accepted that aliasing is a dangerous feature in a PL. Usually, you have access to some kind of aliasing (two pointers set to point to the same variable, for example). Some languages rely on aliasing to overcome deficiencies in the language's data abstraction facilities, while other languages greatly restrict aliasing to increase their reliability.

- **Readability and writability** - Readability and writability influence reliability. A program written in a language that does not support natural ways to express the required algorithms will necessarily use unnatural approaches. These unnatural approaches are less likely to be correct for all possible situations. If a program is easier to write, it is more likely to be correct. Programs that are difficult to read are difficult to write and modify.

Cost

The cost of a PL is a function of many of its characteristics:

1) **The cost of training programmers to use the language, which is a function of the simplicity and orthogonality of the language and the experience of the programmers.**
2) **The cost of writing programs in the language. This is a function of the writability of the language.** The cost of training programmers and the cost of writing programs in a language can be significantly reduced in a good programming environment.
3) **The cost of compiling programs in the language**.
4) **The cost of executing programs written in a language is greatly influenced by that language's design.** A language that requires many runtime type checks will prohibit fast code execution, regardless of the quality of the compiler. Execution efficiency is currently not as important as it used to be (computers are a lot faster today than they were 50 years ago). **Optimization refers to the techniques that compilers may use to decrease the size and/or increase the execution speed of the code they produce.**
5) **The cost of the language implementation system.** One of the factors that explains the rapid acceptance of Java is that free compiler.interpreter systems became available for it soon after its design was released. **A language whose implementation system is either expensive or runs only on expensive hardware will have a much smaller chance of becoming widely used.**
6) **The cost of poor reliability. If the software fails in a critical system, such as a nuclear power plant, the cost could be very high.**
7) **The cost of maintaining programs.** This relies primarily on readability.

The most important of these 7 characteristics are **program development, maintenance, and reliability costs**.

Portability is another criteria that could be used to evaluate PLs. Additionally, there are **generality (applicability to a wide range of applications)** and **well-definedness (completeness and precision of the language's official defining document)**.

1.4 - Influences On Language Design

There are several factors other than the criteria above that influence the design of PLs. The most important of these are **computer architecture** and **programming design methodologies**.

Computer Architecture

Most of the popular languages of the last 60 years have been designed around the prevalent computer architecture called the **von Neumann architecture**. These languages are called **imperative** languages. In a von Neumann computer, data and programs are stored in the same memory. The CPU is separate from the memory, so instructions and data must be transmitted from the memory to the CPU.

The execution of a machine code program on a von Neumann architecture computer occurs in a process called the **fetch-execute cycle**. There's a bunch of extra shit in the book on this section (p. 42) but I don't think it's important.

Program Design Methodologies

In the 1970s began a shift from procedure-oriented to data-oriented program design methodologies. Data-oriented methods emphasize data design, focusing on the use of abstract data types to solve problems. To use data abstraction effectively, it must be supported by the languages used for implementation. The first language to provide support was SIMULA 67. The most recent step in the evolution of data-oriented software development is object-oriented design. This involves data abstraction, which encapsulates processing with data objects and controls access to data, and adds inheritance and dynamic method binding.

Procedure-oriented programming is the opposite of data-oriented programming. That is not to say procedure-oriented programming is dead. It is used especially in the area of concurrency.

1.5 - Language Categories

PLs are often categorized in to 4 grousp: **imperative, functional, logic, and object oriented**. Some people refer to scripting languages as a separate category of PL, but languages in this category are bound together more by their implementation method (partial or full interpretation) rather than by a common language design.

A logic programming language is an example of a rule-based language. In an imperative language, an algorithm is specified in great detail, and the specific order of execution of the instructions or statements must be included. In a rule-based language, however, rules are specified in no particular order, and the language implementation system must choose an order in which the rules are used to produced the desired result.

Recently, a new category of languages has emerged, known as the **markup/programming hybrid** languages. Markup languages are not programming languages, but some programming capability has crept into some extensions of HTML and XML (Java Server Pages Standard Tag Library and eXtensible Stylesheet Language Transformations). They are basically just ghetto half-programming languages.

1.6 - Language Design Trade-Offs

The criteria outlined in section 1.3 are self-contradictory. You can't make a PL which gets full marks in each criteria. Two obviously contradictory criteria are reliability and cost of execution.

The conflict between writability and reliability is a common one in language design. The pointers of C++ can be manipulated in a variety of ways, but they lead to unreliabilities, which is why they are not included in Java.

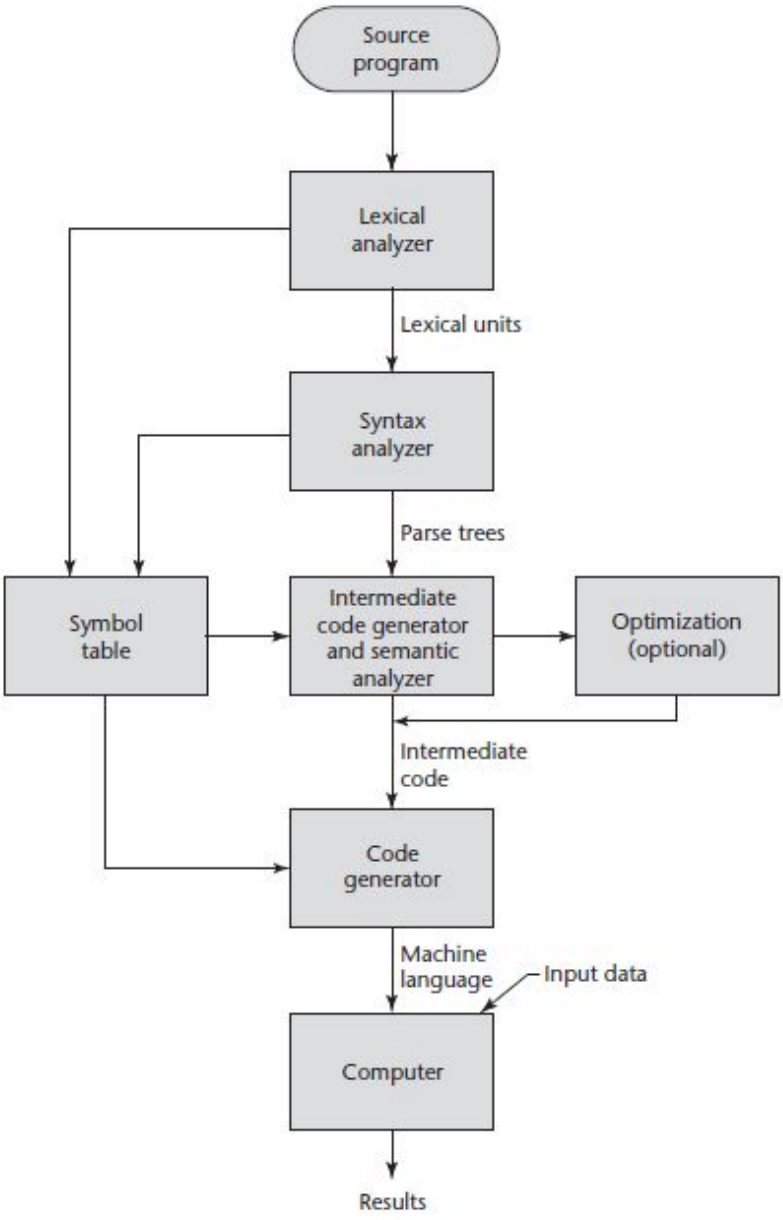
1.7 - Implementation Methods

Two of the primary components of a computer are its internal memory and its processor. The internal memory is used to store programs and data. The processor is a collection of circuits that provides a realization of a set of primitive operations, or machine instructions, such as those for arithmetic and logic operations.

A language implementation system cannot be the only software on a computer. Also required is a large collection of programs, called the operating system, which supplies higher-level primitives than those of the machine language. These primitives provide system resource management, I/O operations, a file management system, text and/or program editors, etc. There's a bunch of shit in the section that I think is not important. Pg 46.

Compilation

Compiler implementation is when programs are translated into machine language, which can be executed directly on the computer. It results in very fast program execution once the compilation is complete. **The language that a compiler** translates is called the source language. The process of compilation and program execution is outlined in the image below. The **lexical analyzer** gathers the characteristics of the source program into lexical units (identifiers, special words, operators, punctuation symbols). The lexical analyzer ignores comments. The **syntax analyzer t**akes the lexical units and uses them to construct hierarchical structures called **parse trees**. These parse trees represent the syntactic structure of the program. The **intermediate code generator** produces a program in a different language, at an intermediate level between the source and the final output of the compiler. The **semantic analyzer** checks for errors that can't be found during syntax analysis. The **code generator t**ranslates the intermediate code to machine language**.** The **symbol table is a database for the compilation process.**



The speed of a computer is often determined by the connection between a computer's memory and its processor, since instructions often can be executed baster than they can be moved to the processor for execution. This connection is called the **von Neumann bottleneck** and is the primary limiting factor in the speed of the von Neumann architecture computers. It is one of the primary motivations for research and development of parallel computers.

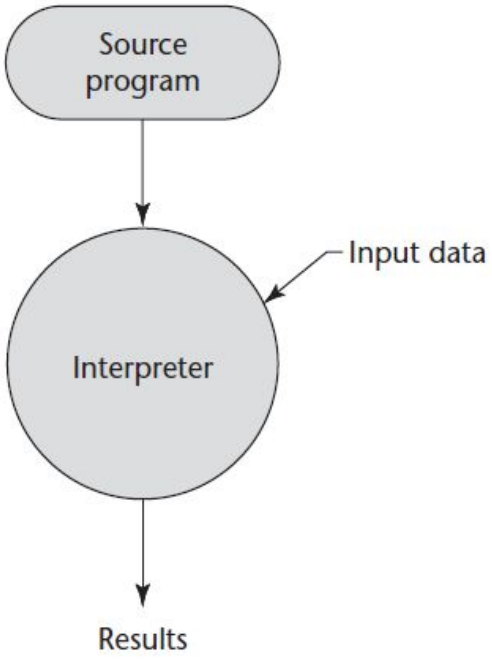
Scroll down for more notes

Pure Interpretation

This is the second implementation method. It is at the opposite end of compilation. With pure interpretation, **programs are interpreted by another program called an interpreter, with no translation whatsoever**. The interpreter acts as a software simulation of a machine whose fetch-execute cycle deals with high-level language program statements rather than machine instructions.

The advantage of pure interpretation is that it allows for easy implementation of many source-level debugging operations, since all run-time error messages can refer to sour-level units. For example, if an array index is found to be out of range, the error message can easily indicate the source line of the errorand the name of the array.

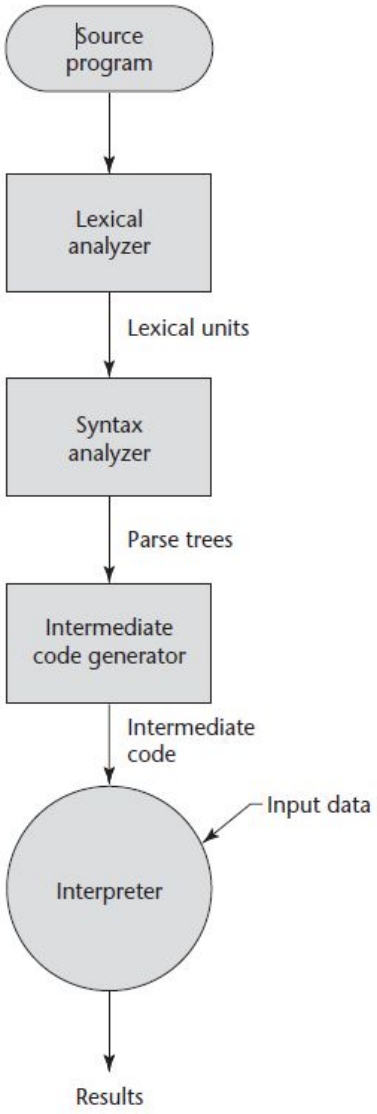
The serious disadvantage of this implementation is that execution is 10 to 100 times slower than in compiled systems. This is mainly due to the decoding of the high-level language statements. Also, regardless of how many times a statement is executed, it must be decoded every time. Thus, statement decoding is the bottleneck of a pure interpreter. Another disadvantage is that is often requires more space.



Hybrid Implementation Systems

The last kind of implementation system is a compromise between compilers and pure interpreters. **Hybrid implementation systems** translate high-level language programs to an intermediate language designed for easy interpretation. This method is faster than pure interpretation. One example of a language that uses this system is Perl.

A Just-in-Time (JIT) implementation system initially translates programs to an intermediate language. Then, during execution, ti compiles intermediate language methods into machine code when they are called.



Preprocessors

A preprocessor is a program that processes a program just before the program is compiled. Preprocessor instructions are embedded in programs.

1.8 - Programming Environments

A programming environment is the collection of tools used in the development of software. This may be a file system, a text editor, a linker, and a compiler. Or it may include a large collection of integrated tools. This section just talks about a bunch of IDEs that you already know about. Pg. 53.

January 22, 2018 (class notes)

"I had a running compiler and nobody would touch it… they carefully told me, computers could only do arithmetic, they couldn't do programs"
-Grace Hopper

Year 1900
Hilbert's 23 problems, one of which was "Can you design an algorithm that can decide if a given polynomial (in many/multiple variables, coefficients are integers) has a root in integers?"
Can you assign each variable an integer so the value = 0? After 70 years, it was finally shown that no such algorithm exists.

1930s
The "Halting problem"
Design a program such that:

Input: python program and this program's input
Output: Determine if the program contains an infinite loop (ie does it halt or not)?

As it turns out, no such algorithm exists, without essentially simulating the program yourself, in which case you might as well just run the original program. (this is expanded upon in **CSC324 - Theory of Computation**)

People thinking about what is computable can be split into 3 main groups:

1) Frege in 1890s: predicate calculus and logic
2) Church in 1930s: lambda calculus (looked at from the viewpoint of functions and recursion)
3) Turing in 1930s: Turing Machines (simple, logical model of what a computer can do).

Church-Turing thesis: The Church and Turing schools of thought are equivalent in terms of what they can accomplish.
Turing complete refers to the things that a turing machine can do. Basically, if your PL can do the following things, it is essentially equal to other PLs that can do these things (can do anything, essentially):

- Sequencing
- Conditional branching
- loop/recursion/branch

Four Main Families of PLs
They can be seen as intellectual descendants of the 3 schools of thought from above:

1) Imperative
    a) Program is a sequence of commands for directing control, I/O, doing calculations, etc

    b) Descendant of the Turing Machine model
    c) Lots of languages here: ALGOL, Fortran, C, Python
2) Object-Oriented
    a) Program is a number of classes
    b) Class: specifies data being represented, and allowed operations on the data
    c) Running the program: A bunch of interacting objects (instance of a class)
    d) Things like Java, C++, C#, Ruby, Smalltalk, Simula
3) Functional
    a) Descendant of the Church development (lambda calculus)
    b) Program is a bunch of function definitions
    c) Running the program: Nested function calls
    d) Favors recursion rather than iteration
    e) There are little "side effects": variables are immutable, so they don't get changed often. Inducing side effects is sort of like "cheating" in functional programming. Loops need an index variable that changes with each loop, but recursion doesn't
    f) Includes LISP, Haskell, SML (LUL)
4) Logic or Rule-based
    a) Descendent of Frege and predicate calculus
    b) Program is some axioms (initial, known facts/things such as 0! = 1, etc) and some set of rules
    c) Running the program: Apply the rules and the axioms to your initial data
    d) Includes things like Prolog, Mathematica

<u>Compilers and Interpreters</u>

- **Compiler: A program that translates a high-level language (a PL you would write in) to a lower level language (this is often machine code, but not always)**
    - Src -> compiler -> Object code -> linker -> execution
    - When the program is run, you don't need the compiler anymore, nor do you need the source.
- **Interpreter: A program that takes source code as the input, translates it to machine language, and executes it.**
    - Need the source code to run when using an interpreter
    - Reads line by line and executes it, and then beings working on translating the next line
    - An advantage is that the code can, in some sense, change itself and the interpreter would adjust accordingly

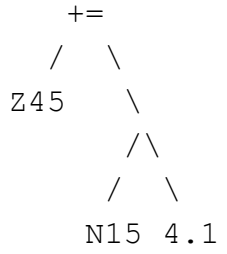
- Compiled languages (speed)
    - Fortran, C, ...

- Interpreted languages (flexibility)
    - Basic, LISP
- Variations:
    - Original BASIC - line by line interpreting and executing
    - Perl has a front-end compiler that scans the whole program for any syntax errors, and then executes the program
    - Java has a front-end compiler that takes source and turns it into **bytecode**. Bytecode is a machine-independent machine code (like a low-level language). On the back-end there is a bytecode interpreter in the JVM, or maybe a compiler ("Just-in-time").

Phases of Compilation

"Front-end" of the compiler

1) **Preprocessor**
    a) Strip the comments
    b) Expand macros or definitions
    c) Including other declarations of library items (such as using a `sqrt()` function from a library)
    d) Conditional compilation
2) **Scanner** or **Tokenizer**
    a) First syntactic check. Turns the program into a sequence of tokens: names, operators, etc. For example, it determines how to interpret
    `x+++y -?-> (x++)+y`
    `Z45 += n154.1+\j`
    b) Produces a token stream
3) **Parser**
    a) Produce syntax tree
    b) Requires knowledge of the grammar



4) **Semantic Analysis**
    a) Type checking
    b) Binding names to definitions
    c) Deals with finding **meanings**
    d) Checking return values from functions

"Back-end" of the compiler

This is where code is actually generated. Is not discussed in great depth in this course, btw. These are covered in a compilers course, usually.

1) **Intermediate code generation**
2) **Optimizations**
3) **Target code optimization**
4) **Machine-specific optimization**

January 24, 2018

"C has the power of assembly language and convenience of ... assembly language."
-Dennis Ritchie

Homework note: have 1 array for tracking if a column is occupied or not:
[2,4,6,-1,-1] etc, where the index is the location the queen is on in that column

C Programming Language

Introduction and History

- 1970s: AT&T and others formed a partnership together to put together a new OS for their companies. This OS was called **MULTICS**.
    - AT&T employees Ken Thompson and Dennis Ritchie then created a spin-off OS, called **Unix**.
    - Ritchie then made the **C programming language**, which was for writing Unix.

Influences

- BCPL: ("Basic Control Programming Language"). This language had 1 type, which was `word`. It had the `=+` operator, and used `{}` for the block marks (`do, end`, etc). The program (compiler) fit in 16 kb.
- B: This language came along and had the operators `++, --, ==`
- C: B with types, pretty much. It is weakly typed
    - K&R C - 1978
    - ANSI C - 1989
    - ISO C - 1999. Aka C
    - C - 1999

C is hugely influential to other PLs like Objective C, C++, C#, and Java

Types

The types in C are:

- `int`

- `float`
- `double`
- `char` - You can do artithmetic on chars, and it will use the ASCII value
- `(void)`

The type modifiers are sometimes considered their own types. They are:

- `short` - For example, you can have a `short int`
- `long` - You can have a `long double`
- `signed`
- `Unsigned`

For unsigned `ints`, they range from 0 to 2^{32} - 1.
There is an operator `sizeof()`, which returns the size required for the given type:

```
sizeof(int)
     //returns 4
```

Arrays

Index starts at 0
When the size is known at compile time:

```
     int a[10];
```

When the array size is dynamic:

We will need to use pointers to understand dynamic array:

```
     a[i] ⟷ *(a+i) //These have the same meaning
     //What we actually do is the following:
     int*a = (int*)malloc(n*sizeof(int)) //cast as int
```

In the code above, `n*sizeof(int)` is the number of bytes, where `n` is some inputted number (such as from a user, like in N-Queens).

When you're done using this array `a`:

```
     free(a) //you need to free up the memory again after
allocating it
```

Strings

A string is an array of `chars`. There is no `string` type. A string is represented as a null-terminated array of `chars`. **Null is a has a special character: `\0`**

```
     char name[4] = "Bob"; //It's 4, not 3, because of the
null-terminator char
     //Most people would write:
     char name[] = "Bob";
```

There is an import call: `#include <string.h>`. It includes functions like:

```
strlen()
strcat()
```

Comments

Originally , the comment format was only `/* … */`. Later, they added support for `//…`

Scroll down

Expressions and Operators

`+=, *=, ++, ==, <, <=, ==, !=, &&, ||, |, ?:`

Reminder:

```
x = (condition) ? <result 1> : <result 2>;
```

There are also bitwise operators, but we probably won't use them in this course. They include:

`&, |, ~, ^, <<, >>`

Control

This is just like the control operators in Java

`if, switch, for, do, while, return, break, continue,` **`goto`**

Don't use `goto` (but you knew this already). C++ has `goto` although nobody uses it.

Pointers

This is where things start to get a bit strange in comparison to other languages. In C, you can get the memory address of anything you want. This is very useful for memory allocation (`malloc`). You can get the address with the unary `&` (unary meaning there's only one operator, or something like that? Google it!!)

```
int x = 17;
int* p = &x;
print p; //address of x
print *p; //17
```

Unary *: in direction

"What is the value being pointed to?"

```
*p //returns 17
*p = 23;
print x; //returns 23
```

Everything in C is passed by value, not by reference. So, with pointers, you can simulate the ability to pass by reference:

```
int x;
//print "Enter a value for x: "
getvalue(&x) //gives our function the address of x
//use x
---------Later in the code----------
void getvalue(int *p){
     *p = 25; //this will change the value of x up top
}
```

A common C error:

```
int * p; //no int value
int y = *p;
//leads to a "segmentation fault"
```

January 29, 2018 (Class notes)

"For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of goto statements in the programs they produce"
-Edsger Dijkstra

C Continued

Pointers continued

`Null` value for pointers:

```
int* p = NULL;//preprocessor sees null and replaces it with 0

int a,b; //two ints
int* p, q; //does NOT create two pointers. The * binds with the p and it really means:
int *p,q; //what p points to is an int. It makes one pointer.
int *p,*q; //creates two pointers
```

Suppose `c` is an array of `int`s

`C[i] ⟷ *(c+i) //this is legal, and C doesn't care about the size of c, so if you go out of bounds, C will just grab whatever is at the byte specified from the c+i position. This can be really bad if it's information in your code, which means it still compiles but creates confusing bugs.` **`There is no range checker in C`**

I/O

Remember to have `#include<stdio.h>`

Output: `printf(string)`

```
printf(string-with-codes, expression);
printf("The answer is %d\n", ans);
```

Input: `scanf(codes, addresses)`

```
printf("Enter n: ");
int n;
scanf("%d", &n);
```

Functions

Basic format of a function definition in C:

```
double hype(double x, double y){
     return sqrt(x*x + y*y); //would need to include
#include<math.h> up at the top of the code file. This is a
function declaration (what?)
}
```

To declare `hyp`:

```
double hype(double, double);
```

Code Organization

Assume single source file for the moment. Overall, your code should look like this:

```
includes
declare auxiliary (everything but main()) functions
main function definition
auxiliary function definitions
```

You need to declare functions before `main` so that when the functions get called in `main`, the program knows to jump down to the function definitions to see what the functions do.

```
int main(){
     [
          Body of code
     ]
     return 0; //this is the same as returning void
}
```

Inside the `main ()`, you may see something like `int argc, char** argu.c` indicates the amount of arguments coming in, and `u` is a pointer to a pointer of a `char.` You don't need these in homework 1, apparently.

Globals

You can declare variables outside of functions, if you want. But you shouldn't do it, because it's bad practice. It pollutes the global namespace, and it's a bad habit to get into. It can especially create issues when you have multiple src files. Instead, you want to declare variables inside of functions and use parameters with pointers to get around it.

However, starting with ANSI C, we have an exception in the case of global constants (such as pi). This is done with `const double Pi = 3.14159.` You could also do this the old school way, with macros (thanks, Jimmy): `#define Pi = 3.14159.`

Structures

There are no classes in C. However, they do have structures, which you can view as very rudimentary classes (holds data, but has no methods and no regulation for access control → automatically public). Structures are a way to collect data values in one variable. For example:

```
struct complex{
     double re;
     double im;
};
struct complex z;
z.real = 4.3;
z.im = -2.9;
struct complex* p = &z; //pointer to a complex
p.im //THIS IS ILLEGAL. DOES NOT COMPILE
(*p).im //This is technically legal, but a bit awkward. Instead:
p -> im //This is equivalent to the line above. This is known as
syntactic sugar
```

Semantics

- No `boolean` type (before C99)
    - So, how did they do `if(cond) then result` ? The answer is that they used ints to represent true and false, **so conditions will evaluate to numbers**. In conditions, **0** means `false,` and **nonzero** means `true.` So, say `p` is a pointer:

```
if (p) {...} //if p points to NULL (0), then the
expression won't evaluate. If p points to a value, it will
evaluate.
```

Other Keywords of Interest

`register:` Suggest this variable should be stored in a register. Is good for things we need lots of quick access to, such as an index tracker:

```
for (register int = 0; ...)
```

`union:` Do not use this. This was for back in the day when memory was very sparse. `union` let you use a spot for two different things at once (give it a name for an `int` and a `float`).

Homework: Read chapter 2

January 31, 2018 (Class notes)

"Colorless green ideas sleep furiously"
-Noam Chomsky

Reminder, during the compilation process, we have the subprocesses: scanner → parser → semantic analysis. The scanner and the parser are on the syntactic side of things.

Formal grammars (See CSC 324 for more info)

Noam Chomsky is the founding father of **formal grammar** (also known as **transformational grammars**. His PhD thesis where he developed this was in 1955).

We need some definitions:

- **Language - A set of valid strings constructed from an underlying alphabet**
- **Grammar - Describes rules for forming valid strings in a language**
-

Formal Grammar consists of:

1) Set of "terminals" (characters, keywords, building blocks, etc)
2) Set of variables
3) Set of production rules showing how to make substitutions (eg: prepositional phrase → preposition noun)
4) Start variable

A string *s* is in the language generated by a grammar *G* if you can manufacture *s* using the rules of *G*, starting from its start variable: $s \in L(G)$

Eg: Terminals: {a, b}

Variables: *S*

Rules: $S \rightarrow aSb \mid \in$

So:

$S => \in$

$S => aSb => a \in b = ab$

$S => aSb => aaSbb => aabb$

$L(G) = \{a^n b^n : n >= 0\}$

Chomsky Hierarchy

He said that languages have different levels of complexity. We have very simple languages, or very complex languages. He said there are 4 hierarchies of languages. There are 2 ways to look at this, being the **formal language view** and the **machine view (formal computer)**.

The formal language view is:

1) **Regular languages (scanners side)**
   a) Has a grammar where all the rules have the form A → a (variable → terminal, like *S=> ∈)* or A→ Ba (variable → variable)
   b) Ex: Terminals: {0, 1}
      Rules: S → S0 | S1 | ∈
      S => S1 => S01 => S001 => …

      Language: all binary strings
2) **Context Free languages (parsers side)**
   a) Has a grammar where every rule has the form: A → α (left side is a variable, but right side is **any** combination of variables and terminals [a superset])
   b) You can further reduce this to special cases: All rules have the form A → BC or A → a
3) **Context-Sensitive languages**
   a) Rules have the form αAβ → αγβ
      Ex: 0A0B → 01110B
      |A|B → 0111B
4) **Recursively enumerable languages**
   a) Rules are unrestricted: α → β

In the machine view, each level corresponds to a particular model of computation/formal machine:

1) **Finite automaton (state machine)**
2) **Pushdown automaton (stack, because you need more memory)**
3) **Linear bounded automaton**
4) **Turing machine**

Scroll down

Scanners

Study things in level 1 of the machine view (finite automaton) for: variable names, literals.

- **Finite automaton** - Think of this as **a simple model of computation given by a state machine.** Consists of:
    - Finite number of states
    - Transition rules between states
    - One start state
    - Some states are "accept" states

  To determine if an automaton accepts a given string *s*:
    1) Start in the start state
    2) Make transitions based on characters in *s* (one transition per character)
    3) At the end: if you are in an accept state, say yes. Otherwise, say no.

  Ex: Alphabet: {0, 1}

  See the back of your ML notes book to see the example and transfer it to this doc!!!

  L(D) = all binary strings with an odd number of 0s

February 5, 2018 (Class notes)

"From then on, when anything went wrong with a computer, we said it had bugs in it."
-Grace Hopper

Review of last class: Gave an overview of Chomsky Hierarchy/Formal languages.

1) Regular Languages ← → scanners
    a) From a formal language standpoint: A → aB or A→ a
    b) From a finite automaton standpoint (a machine standpoint)
    c) From a regular expression standpoint

Deterministic Finite Automaton

- Finite state machine:
    - Finite number of states
    - 1 of them is a start state
    - Some of them are accept states
    - Transition arrows to define how we move between states on input
- **Deterministic** finite automaton:
    - Each state has exactly **one** outgoing arrow for each input possible input character

Ex: Design a simple automaton for an integer literal. For example, accept things like 47, +34, and -355, but **not** 14. (as in, 14.0 without the 0). See the back of ML notebook!!

Scroll down

Regular Expressions

This is a system for describing languages (such as strings) via patterns. A **regular expression over a given alphabet, Σ, is defined recursively:**

Base cases: ∅, ∈ (empty string) (confirm these 2 from jimmy), any single *a* ∈Σ

Recursive cases: Say R,S are regular expressions over Σ. Then so are:

1) **RS - Juxtaposition. L(RS) = L(R) ○ (LS), where ○ is string concatenation**
2) **R* - Repeat 0 or more times. L(R) = ___ ___ ___, where ___ is something in R and each ___ may be a different group of things in R. So this means the empty string (∈) is always in L(R*).**
3) **R/S - Alternation. L(R|S) = L(R) ∪ L(S)**

So, let Σ = {a, b}. Then,

a|b → a or b

(aa|b)* → ∈, aa, b, bbbbbbaa, aabaaaabb

(b*ab*)(ab*ab*)* → (b*ab*) matches any string with just one a. (ab*ab*) matches any string with exactly two a's and starting with a. Note that with respect to the whole expression, we don't care about the preceder of the second (). So, the whole expression matches: Any string with an odd number of a's. Some examples that match this expression are: bbabaabbaaaab

Regular expressions are equally powerful as deterministic finite automaton (theorem from CSC324).

So, try writing the regex for integer literals, where <digit> means 0|1|2|...|9

(+|-| ∈)<digit><digit>*

(+|-|<digit>)<digit>*

These ALMOST work, but we need richer syntax for regex.

This gives us the ? operator:

R? = R| ∈

You can think of ? as 0 or 1 time

R+ = RR*

+is essentially at least one R

So, the integer literal regex becomes:

(+|-)?<digit>+

Some other, common operators are:

. = any single character

\. = .

[0-9] = any single digit

[a-z A-Z] matches any letter
[abcd] matches a|b|c|d
[^abcd] matches any letter except a, b, c, d

$ = end of line/string
^ = beginning of line/string

February 7, 2018 (Class notes)

"You need the willingness to fail all the time. You have to generate many ideas and then you have to work very hard only to discover that they don't work. And you keep doing that over and over until you find one that does work."
-John Backus

Last class: Discussed regular languages and regular expression and how we need both of these to build scanners (lexical analyzer). To build a scanner:

- Use automata or regex to build a scanner
- Textbook has a nice example in 4.2
- Imagine making a big table-driven scanner
- Maybe you could write a program that takes as an input a description of components of your language (what names are, what literals there are, what key words there are, etc), and then outputs a program: This program would be a scanner for your programming language
    - There are standard unix tools that do this, like lex (flex on macOS). An example of this input file can be seen here: http://www.quut.com/c/ANSI-C-grammar-l.html

Parsing
Concrete syntax - Rules for valid programs using tokens as the underlying alphabet
Expressed by using a context-free grammar (CFG).
Can use Chomsky's transformational grammars:
<statement> → <if statement> | <for loop>
<if statement> → if (<condition>) <statement>

John Backus - 1956ish, working at IBM, developed essentially the same idea as Chomsky, in a cs context.

- Another independent development of CFGs
- Was supplemented by Peter Naur
- Resulted in **"Backus-Naur Form" (BNF or EBNF)**

BNF grammar:

- Variable: Word in angle brackets, like <expression>
- Terminal: string (some token)
- Production: in place of Chomsky's →, they use ::=
- Use | for alternation (or)
- Use <empty> instead of ϵ

EBNF grammar: Added some regex-like syntax

- {stuff} means 0 or more occurences of stuff (similar to *).
- [stuff] means that stuff is optional (is equivalent to saying "stuff or empty")
- Ex: <intLit> ::= <intLit><digit> | <digit> OR <intLit> ::= {<digit>}<digit> where <digit> ::= 0|1|2|...|9

Parse Trees and derivations

Top-down derivation:

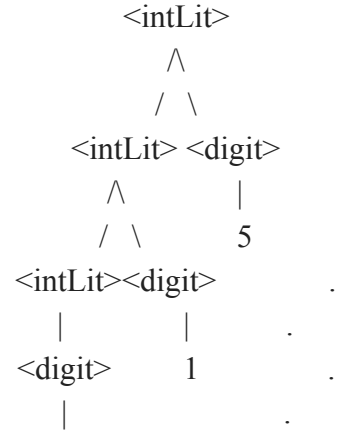
- Start with a variable
- Use productions to generate a string

Ex: make 215

<intLit> ⇒ <intLit><digit>
⇒ <intLit><digit><digit>
⇒ <digit><digit><digit>
⇒ 2<digit><digit>
⇒ 21<digit>
⇒ 215

To summarize the above: <intLit> ⇒* 215 **(SHOULD BE ⇒ WITH STAR ON TOP, CAN'T DO IT ON COMPUTER)**

This derivation is not only top-down but also a **left-most** derivation (always derive the left most thing in the current expression).



Grammar: <intLit>::=<intLit><digit>
This rule is **left recursive**, and thus produces a parse tree with a larger left side

However, if you did <intLit> ::= <digit><intLit> | <digit> so it's **right recursive**. The tree gets mirrored basically, with the 2 on the left, and the tree is right skewed:

<intLit>
/\
<digit><intLit> (finish drawing this shit)

Arithmetic
<expression> ::= <expression>+<variable> |
<expression>-<variable> |
<variable>
<var> ::= a | b | c
Top-down parsing for: a - b + c, left recursive:

Produces (a - b) + c
Top-down parsing for a - b + c, right recursive:

Produces a - (b + c)

A language has a number of operators.
Operators are organized by precedence:
*, /
+, -
Etc...

But we need to also list the operators' associativity:
In an expression involving operators at the same level of precedence, and in the absence of any parentheses, the associativity tells you whether they should be evaluated left-to-right or right-to-left.
Moral: For an operator that associates left-to-right, use a left-recursive definition. This will give you a parse tree with the correct meaning.
Right-to-left associativity:
** (exponential) in Fortran
= in C (chaining =, like a = b = c = 0 sets all variables equal to 0)

Scroll down

Another language definition:
<expression> ::= <expression><operator><expression> |
(<expression>) |
<variable>
<operator> ::= + | - | * | /
<variable> ::= a | b | c

Parse: a + b * c
Come up with 2 different parse trees for that expression with the given grammar:
See back of paper print of CSC250 HW3 (lol)

Graham's soln suggests (a+b)*c
Mitchell's soln suggests a+(b*c)
This is a problem, because this grammar is **ambiguous. A grammar G is ambiguous if there exists a string *s* in L(G) with at least 2 structurally different parse trees.**

February 12, 2018 (class notes)

"One morning I shot an elephant in my pajamas. How he got in my pajamas I'll never know."
-Groucho Marx

Grammar rules for arithmetic expressions

- Need one rule per precedence level
- Want the **low precedence operators** to be **high** in the parse tree
- So <expr> should expand to the **lowest precedence operators**
- Then, subsequent rules work in higher and higher precedence operators

This way, you'll get the right parse tree

So, improved grammar based off the last grammar mentioned in last class (scroll up!) is:
<expression> ::= <expression> + <term> | <expression> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= (<expression>) | <variable> | <intLiteral>

So, again, lets parse a + b * c:

<expression>
/\
<expression> + <term>
| /|\
<term> <term>*<factor>

<factor> Factor> <var>
<var> <variable> c
A b

This is the **concrete syntax tree**. Ask Jimmy for the abstract syntax tree

<u>if statements</u>

```
<ifstmt> ::= if (<expr>) then <stmt> |
              If (<expr>) then <stmt> else <stmt>
<stmt> ::= <ifstmt> | <expr> | <whileloop> | …
```

So, lets parse the following: `if (a) then if (b) then c else d`
Check 250 HW3
It has 2 different parsings: does the `else` attach to the first `then`, or the second?
Note that this is an ambiguous grammar, too!

There are several solutions to this ambiguity:

1) **C solution:** Leave the grammar ambiguous. The meaning still has to be nailed down, so in C, the else binds with the **closest** `if` in the absence of braces.
2) **Java solution:** Fix the grammar so there is no ambiguity:
   ```
   <ifthen> ::= if (<expr>) <stmt>
   <ifthenelse> ::= if (<expr>) <stmtNoShortIf>
                    else <stmt>
   ```
3) **Algol solution:** Mandate {} or being end

<u>February 14, 2018 (class notes)</u>

"John while Jack had had had had had had had had had had had a better effect on the teacher"
-Hans Reichenbach

```
<ifstat> ::= if (<expr>) then <stmt> |      (this one is ambiguous)
             If (<expr>) then <stmt> else <stmt>
<stmt> ::= <ifstat> | <expr> | <???> | …
```

```
<ifthen> ::= if (<expr>) then <stmt>     (this one is not ambiguous)
<ifthenelse> ::= if (<expr>) then <stmtNoShortIf>
                           else <stmt>
<stmt> ::= …
<stmtNoShortIf> ::= … (omit <ifthen>)
```

Scroll down!

Check back of CSC250 notebook page -2!

Ways to handle this:

1) **C way:** Leave the ambiguity in the grammar and specify the meaning elsewhere
2) **Java way:** Fix the grammar so only one parsing is legal
3) **Mandate blocks (Algol way):** Beginning/end of `then`, `else` parts must be marked. Can be seen in Pascal and Maple, too. So, `begin ... end`.

Suppose you opt for choice #3:

```
if () then
     _____
else
     if () then
          _____
     else
          if () then
               _____
          else
               _____
          end
     end
end
```

It gives you an ugly chain of indents and `ends`. You can fix this by creating the `elseif` keyword, and making it a more complicated `if`, such as `elif` or `elsif`

```
if () then
     _____
elif () then
     _____
elif () then
     _____
else
     _____
end
```

Scroll down

Parsing Algorithms

In general: given a CFG (context free grammar), there is a parsing algorithm that can determine if a given string is in the grammar in $O(n^3)$ time.

Of course you would be much happier with O(n) time.

Certain special grammars allow this. From CSC 324, these are **deterministic CFGs.**

Two kinds in particular: LL grammars and LR grammars (named after corresponding parsers). The first "L" says to read the program left-to-right, and never back up: O(n). The second "L" or "R" corresponds to the kind of derivation: leftmost or rightmost.

LL: Top-down, leftmost derivation (like what we've done in class). For example:

S → aAc, A → aA|b

Top-down parse of aabc: A => aAc => aaAc => aabc

LR: Bottom-up derivation

aabc <= aaAc <= aAc <= S

1) **LL (or LL(k)** where k = number of tokens of lookahead you can use. Often, k = 1**)**
   a) Use EBNF rules to design parser
   b) Use your lookahead token to decide what rule to use
   c) So, your grammar needs to be in a special form (to be able to tell what's going to happen with only a little bit of lookahead).
      i) A → A + B this left recursion is **bad**
      ii) <var> → <id> | <id>[<expr>] this is also **bad**
      iii) Fixed with: <var> → <id><A>
      <A> → epsilon | [<expr>]

Read the rest of chapter 4: 4.3-4.5

2) **LR**
   a) Was invented by Donald Knuth
   b) LR parser: like a finite automaton that has a stack, which lets you remember previous states or things you've seen. This kind of automaton is known as a **PDA**.
   c) Referred to as "shift-reduce" parsers:
      i) "Shift" - Push something onto the stack and go to a new state
      ii) "Reduce" - Pop the stack (RHA of rule) and push the variable on the left side. Then advance the state

Given a grammar (right form - DCFG. LL <= LR (what???)), you can produce a state diagram for a PDA. This can be automated (and has been!). Common tools are yacc and bison. Most people use one of these to make the parser for their language.

February 19, 2018 (class notes)

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows your whole leg off."
-Bjarne Stroustrup

C++

Designed by B. Stroustrup. After his PhD he went to Bell Labs and developed C++ in the early-mid 80s. He liked C, but the idea of OOP only really existed in the world of academics. So, the influences of C++ were C (exprs, ctl), Simula (oop, classes), and Algol (type checking).

Design Goals

1) Speed
2) Object-oriented
3) Extends C (C++ has to compile anything from C. So, for example, this is why it has a `goto` statement)

Here's some C++ code, because why not

```
#include <iostream>

using namespace std;

int main(){
     int b[10]; //static array, or stack-based array.

     //dynamically allocated array, or heap based:
     int n;
     count << "Size? ";
     cin >> n;
     int* a = new int[n]; //new allocated memory dynamically and
returns a pointer

     //Every new should have a corresponding delete
     delete [] a; //array delete. Note that it deletes the stuff
at the address of a, not the pointer a. a is automatically deleted
at the end of the routine (like the end of main() or the function
it's in)

     //In C++ you can dynamically allocate anything you want
     int* p = new int;
```

```
    *p = 43; //note there's not really a reason to do this

    delete p; //regular (non-array) delete, so no []

    cout << "Hello world!" < endl;
    int x;
    cout << "What is x? ";
    cin >> x;
    cout << "x is " << x << endl;
    return 0
}
```

For HW2, put `using namespace std` at the top. You need to do that to get the contents of std into the global namespace. Otherwise, you need to do `std::cout`.

I/O

Not like C at all.

- `#include <iostream>`
- `cout` → `<<` console output
- `cin` → `>>` console input

Types, control

Just like C. However, it also adds a `bool` type, but it's still just a 0 or 1 internally. Conditions still evaluate in a numeric context (0, 1, etc).

Something new in C++, not in C:

References

These are **crucial**.

- A reference allows you to have two names for the same variable/object
  - `int x = 10;`
  - `int& r = x; //r is another name for x`
  - `r++;`
  - `cout << x; //11`

So, why do we want to do this? Some applications are:

1) Pass by reference

```
void f(int& r){
     r++; //in main x is some value and r refers to it
}
```

2) Passing objects to functions efficiently

```
#include <string>
string s = "Bob";
f(s);

void f(const string& name){ ... } //note that const is a
promise that we won't change name. Allows you to prevent
manipulating name inside f(). Compiler will catch these
errors if you break the const rule.
```

Example of throwing an exception in C++:

February 21, 2018 (class)

"I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I Can no longer figure out how to use my telephone."
-Bjarne Stroustrup

HW2 question 1 answer:
<switchStmt> ::= switch (<expr>) \{
{case <constExpr>: {<statement>}}
[default: {<statement>}]
{case <constExpr>: {<statement>}}
\}

Classes in C++
Creating objects
Eg: date object
In the client, you can do `Date d(2, 21, 2018);`
Note that you can create objects just like you create primitives. You don't need the `new` keyword like in java.
But, you can also do `Date * pd = new Date(2, 22, 2018);` which creates a date on the heap. Of course, you would then delete these once you're done with them.

Defining Classes
You typically have **two** files per class.

1) Class definition file:
   File would be called `date.h`
   ```
   class Date {
        public: ...
   ```

```
        //constructors, public methods, interface and how you
use the class
        Date(int, int, int);
        int compare(const Date&) const;
        ...
        Private:
        //fields, private, ancillary methods
        int m, d, y;
        …
    };
```

- Typically, **only declarations** of methods go here

2) Class Implementation file.

   This would be called `date.cpp`. The file looks like:

```
#include "date.h"
//method definitions
Date::Date(int mm, int dd, int yy){
     m = mm;
     d = dd;
     y = yy;
}
int Date::compare(const Date& otherDate)const{
     //int is the return type, and Date is the class
     ...
}
```

Note that `this` keyword is a pointer, so you would need to do something like `this->m` to get this object's month.

In C++, when you write a method, ask if this method could possibly alter a field in this current object. If it doesn't alter anything, declare it const.

<u>Overloading Operators in C++</u>

There are three ways to overload an operator in C++:

1) As a member function (method). This is the most common way to overload.

```
Rational q(1, 2);
Rational r(3, 4);
Rational s = q + r;
s = q.operator + (r);
//declaration is as follows:
const Rational operator+ (const Rational&)const
```

```
    //first const above tells the compiler that the return value
is not assignable (?). Prevents (q+r) = t; for example
```

2) As a nonmember function

```
    s = q + r; //binary
    s = operator+(q, r) //two args
```

This is not a method of the Rational class, so it cannot see the private field variables of Rational objects (numerator and denominator). So why would we want to use this? Here's an example, using *= and *

```
    class Rational { //do *= as a method and * as a nonmethod
        public:
        ...
        Rational& operator *= (const Rational*);
        private:
        ...
    };
    Rational operator* (const Rational&, const Rational&);
```

Implementation file:

```
    Rational& Rational::operator*= (const Rational& r){
        num *= r.num;
        den *= r.den;
        simplify();
        return *this; //return what this points to
    }
    const Rational operator*(const Rational& q, const Rational&
    r){
        Rational ans = q;
        ans *= r;
        return ans;
    }
```

The client may want to do int*Rational or Rational*int, so you need a nonmethod overloading for the first statement, int*Rational, since the * always binds to the left. Or, you just do the better thing: *= member, Rational*Rational nonmember (as we did above), **and** write a constructor that takes one int, n, and makes n/1.

3) But what if we want to do `cout << q`? cout is not a class we wrote, so we can't edit it ourselves to define the print method. So we can't use option 1. So it must be option 2, right? **No.** In case 2, we can't see `num` and `den` in `q`. So, the solution is `friend`. Friend can see all the components, private or not. **The only time you use `friend` is in input and output purposes.**

February 26, 2018

"There are only two kinds of languages: the ones people complain about and the ones nobody uses"
-Bjarne Stroustrup

Operator Overloading continued

Recap:

- As a member
  - x=y <-> x.operator=(y)
- As nonmember
  - x+y <-> operator+(x, y)
- As friend
  - <<, >>

Overloading Functions in general

Is allowed in java, C++. **Not** allowed in C. In general, when you overload a function, you have given it more than one meaning.

Each definition must differ in:

- Number of parameters, or
- Type of parameters, or
- `const` method or not (In C++)

Ex: `f(int), f(int, int), f(const string)`

But can you overload on return type?

```
int f(int)
double f(int)
```

No. The compiler can't tell which function to use

Ex:

Myclass

```
public:
     int f() const;
     int f();
```

Client

```
const Myclass obj1;
Myclass obj2;
```

```
obj1.f(); //const
obj2.f(); //non-const
```

Overloading brackets

Let's use the Rational class as an example.

Client:

```
Rational r(5, 7);
int n = r[0]; //want the value of the numerator. Return int. No changes to the object, so it's const.
r[0] = 8; //want the location of the numerator. Return int&. This one alters the object, so it's not const.
```

Member declarations:

```
const int operator [] (int) const;
int& operator [] (int);
```

Function bodies (same for both functions):

```
Rational::=operator[] (int k) {
     if (k == 0) return num;
     if (k == 1) return den;
     throw out_of_range("invalid index");
}
```

Miscellaneous comments on operator overloading

- Assignment operator: check for assignment to self

  `op = (const Myclass& rhs)`

  Body:

  ```
  if (this != &rhs){
       //do the copying
  }
  return *this;
  ```

- General rules on operator overloading in C++
  - Can't make up your own operator
  - Can only overload on object types
    - So you can't define `float % float` because in C++, % only accepts integers
  - Can't change the "arity" (number of arguments, such as a unary or a binary operator)
    - Ex: You can't do `q!p` because ! is unary only
  - A few operators can't be overloaded in C++:
    - ?:

- .

Note that there is **no** operator overloading in Java because:

- Makes code less transparent
- Hard to get right (the goal is to get your overloaded operators to work like they do on primitive types)

Some other useful operators to overload:

- Typecasting (think casting a rational to a double, for easy conversion/representation)
- Parentheses
    - For example, if you want to save some state between calls. For example, you call `f(100)`, and then a later call to `f(150)` is easier
      Solution: make a class `F`
      Make object `f`
      `f.go(100)`
      `f.go(150)`
      In C++, the `go` can be replaced with `operator()(int),` becomes:

`f.operator(100)`

Why do we want to overload these operators? Because we often want to write our definition of the operator **once** for any type. For example, for a method that returns the absolute value of a number, it's okay for a Rational number if we support `Rational < int` and `-Rational` (unary -, indicate negative).

In C++, the mechanism for generic programming described above is **templates**.
Two cases: **function templates** and **class templates**.

```
template <class T> //(instead of class, you can write typename)
T abs(const T& n){
     if (n < 0) return -n;
     return n;
}
```

February 28, 2018 (Class notes)

"Your quote here"
"If you think it's simple, then you have misunderstood the problem"
-Bjarne Stroustrup

C++ Wrap-up

- References
    - A reference always refers to something. **NO NULL REFERENCES**. Doesn't change over its lifetime; no idea of a null reference.
      `int &s; //this is illegal`
- Casting
    - Suppose a, b are of int type, and we have `double x = a/b; //int. Division,`
    - Common solution is to cast `a` to double first, if we were in some other language like java.
    - In C++, there are 4 different ways to cast types (compatible with C):
        - `double x = (double) a / b; //original c style`
        - `double x = double(a) / b; //java style`
        - `double x = static_cast<double>(a) / b;`
    - Why this last cast? In the case of inheritance, we might have some object with a slightly different type
- Overloading casting

  `Rational:` want to be able to cast to double

  Declaration:

```
operator double() const; //notice no return type, because it's implicit that we return a double.
```

  Definition:

```
Rational::operator double() const{
     return static_cast<double>(num)/denom;
}
```

  Client:

```
Rational r(3, 7);
cout << static_Cast<double>(r);
```

    - As mentioned, there was no return type. Just like constructors, and the next item to be discussed
- Destructor
    - Fundamental feature of c++ that is not the same as java. Recall the memory management of heap vars (new) is up to the programmer to manage.
    - But what if a class constructor uses `new` to dynamically allocate some memory?
    - The solution in that case is the special method called the destructor.
    - Destructor gets called automatically when an object is destroyed
    - The name of the destructor is `~ClassName()`
- Class Templates

        - For example, suppose you write a class for a stack of ints. But suppose we also want to write a class for a stack of doubles, or strings? That would be annoying, having to define the class for each different type
        - With a class template, it's like writing a generic class. The type is a parameter
- Default methods
    - C++ automatically supplies a few methods for us even if we don't ask for them.
        - op =
        - Copy constructor
        - no-arg constructor, if there are no constructors (would be provided if you defined a class with no constructors in it, but that would be a weird thing to do in the first place)
        - You can disable these by declaring them as private methods in your .h file, but then **not** defining them.
- Standard template library (STL)
    - Includes lots of templates for all your favorite container classes:
        - Stack
        - Queue
        - List
        - forward_list
        - Set (implemented with trees)
        - Multiset
        - Deque (double ended queue)
        - Map
        - Vector (like an array list)
- Inheritance
    - Very different approach to inheritance compared to what java does. Will be elaborated on at a later date

That's all for C++, I suppose.
Next up is chapter 5:

Names, binding, and scope.

Binding

Defined as an association between two things
Variable - value
Variable - address
Name - which variable

A **binding time** is when an association is made. Some possible binding times are:

- Language design time

- Language implementation time (think the size of an int. In Java, an int is always 32 bits, but in C it depends on the machine)
- Program writing time. For example: what "x" refers to in java according to where in the code it appears
- Compile time
- Link time (program layout)
- Load time
- Run time
    - At startup
    - Function/block entry
    - Statement execution

Static binding = before runtime
Dynamic binding = during runtime

Static binding promotes error-checking, and thus has less to do at runtime, so it's faster.
Dynamic binding promotes flexibility in exchange for the facets mentioned for static binding.

Names (with meaning)
There are a few categories for names:

- **Reserved words** can't be used for other purposes. It is already meaningful (if, while, switch, etc).
- **Keywords** are already meaningful, but **might** not be reserved. FORTRAN has no reserved words, but it still has keywords. C has 32 reserved words, Java has 50, and C++ has ~90. In COBOL, there are ~500 reserved words.
- **Other predefined words** include things like `string, cout, printf, sqrt`.

You can use the number of reserved and keywords in a language as a sort of measure of the language's complexity.

March 12, 2018 (class notes)

"The use of COBOL cripples the mind; its teaching should, therefore, regarded as a criminal offense"
-Edsger Dijkstra

For the test corrections, only do corrections from problems 2 - 7 (unless you missed less than 15 points in problems 2 - 7).

Scroll down

Variables

This refers to user defined words.

Six attributes of variables:

1) **Name** - **Most** variables have names, but **not all** do.
   - `int *p = new int; //right side of the assignment is a variable without a name. A place to store some int.`
   - Particular rules for name formation. Depends on the language. For example, case-sensitivity, or length restrictions.
2) **Address** - Also referred to the variable's **location**, or **L-value**. Normally bound at load time.
3) **Value** - Referred to as the variable's **R-value** (**right** side of the assign statement).
   - Named constants have a fixed R-value.
4) **Type** - Determines the possible values the variable can store. Binding of the variable to the type can be static or dynamic.
   - **Static** - More common. Something like `int x`. It is more explicit.

     Less commonly: Implicitly static variables. For example, in old Fortran, if the **first** letter of the var name is `i ... n`, then it has integer type. Else, it has real type.

     Some implicit information for types too, in **Perl**.
   - **Dynamic** - Common in scripting languages (Python, SML, Javascript, Perl, PHP). No declaration of types for variables. The value of variables is interpreted in context. Offers flexibility at the price of speed.

In a compiled language, you typically have static typing. This lets you catch more errors by the compiler. Increases runtime speed, since you don't need to do any type checking or conversion.

5) **Lifetime** - Time interval between creation and deletion of variables. If a variable outlives its binding to a name, or any way to access it (such as a pointer), it is deemed **garbage**. Some languages have a garbage collector just for this. Others don't (C, C++), so you must handle this yourself, else, you will have a memory leak!

Conversely, if a binding to memory (a memory location) outlives the variable it refers to, this is **dangling reference**.

Categories of variables with respect to lifetime:

a) **Static variables** - The variable is bound to its memory location at the start of execution. It remains bound for the entire run. Common example: static field in class, which means every object uses the same field. Less commonly, we see a static variable that is global or inside a function. C allows any variable to be called static. The opposite of static is auto, but it is implied if you don't declare something as static, so nobody writes it down. In old Fortran, all variables were static. This means **recursion is impossible in old Fortran**.

b) **Stack dynamic variables (or "stack")** - This is the common variable that we most often work with. With these variables, the storage binding happens at runtime when entering a

function or block. Memory allocation comes from the "program stack". For example: `main` calls `f()` which calls `g()` which calls `g()`. At runtime, we allocate memory to hold the variables of main. When main calls `f()`, we add onto the stack and allocate enough memory to store the variables for function `f()`. Same happens when we reach `g()`. In general, the allocation and destruction are automatic.

c) **Explicit heap-dynamic variables** - "Heap-dynamic" refers to the variable's memory being allocated on the heap instead of the stack. "Explicit" refers to using a word like `malloc` or `new` to declare the variable. For example:
`int *p = new int(4); //left side is stack dynamic, right side is explicit dynamic`

d) **Implicit heap-dynamic variables** - Such as in scripting languages.

6) Scope - The scope of a variable is the collection of statements where that variable is visible (i.e. it can be used). A language has **scoping rules** which tell you what is legal, and how a name resolves. For example: `MyClass` has field `int x`, and the constructor has a parameter `int x`. In the body of the constructor, what happens if I use `x`? In C++, it refers to the parameter `x` (closer one). The two principal scoping strategies are **static scoping** and **dynamic scoping**.

a) **Static scoping (lexical scoping)** - Can resolve scoping at compile time (as far as what each name refers to and whether it's legal). It depends only on how the code is *written*, not on what happened in runtime.
Common rule: Use the "closest" declaration. Note that some languages like Pascal and Ada allow procedures (functions) to be declared within other procedures.

```
main{
      int x;

      g(){
            int x;

            print(x); //refers to g's x
      }

      print(x); //refers to main's x
}
```

A scope is like a block. It is just a place where a variable declaration could occur. Scopes are either **disjoint** or **nested**.

b) **Dynamic scoping** - If not local, check the history of execution leading up to this point and use the **most recent definition.**

March 14, 2018 (class notes)

Typical Implementation Strategy for Static Scoping

- This is one job of the semantic analyzer
- Would make a **symbol table**
- Each block/scope would have a **dictionary** consisting of name/definition pairs.
- Forms a stack of dictionaries at each scope

Example:

```
int h, i;
void B(int w){
     int j, k;
     i = 2*w;
     w = w + 1;
}
void A(int x, int y){
     float i, j;
     B(h);
     i = 3;
}
main(){
     int a, b;
     h=5; a=3; b=2;
     A(a,b);
     B(h)
}
```

Global: <h, 1>, <i, 1>, <A, 8>, <B, 2>, <main, 14>
B: <j, 3>, <k, 3>, <w, 2>
A: <i, 9>, <j, 9>, <x,8>, <y,8>
Main: <a, 15>, <b, 15>

When compiling function B, the stack looks like:

B dictionary
Global dictionary

Sees i, w:

w is the w in the B dictionary
i is the i in the global dictionary

Switching topics now…

Scroll down!

Perl

"The three chief virtues of a programmer are: Laziness, Impatience, and Hubris"
-Larry Wall

Perl dates back to the late 1980s. It is a "scripting" language (automate common system tasks, such as moving files or other command line statements etc). A scripting language is usually made/used with respect to an operating system. It was also fairly commonly used for web applications (now we have ruby, PHP, python). Invented by Larry Wall.
Perl stands for **Practical Extraction & Reporting Language**, as well as **Pathologically Eclectic Rubbish Lister**.

Overall Philosophy

- Practical
- Flexible
- Forgiving

Strengths of Perl include:

- Text processing
- Easily communicates with the system, file i/o

```
#!/usr/bin/perl -w
chmod 755 !$
```

Note that in Perl, ; is not a statement terminator, but rather it is a statement **separator**. You need ; between two statements on the same line, but not two statements on different lines.

Perl Features

- No main
- Variables:
    - Simple variables are **scalar** variables: A number or a string.
        - **Must** begin with `$`
        - No notion of types
- Operators: Like C
    - +, -, +=, =, *, etc
    - ** for exponentiation
    - Strings: `.` for concatenation, `x` for repeat
    - Numbers: ==, !=, <, <=, etc
    - Strings: eq, ne, lt, le, gt, ge

    - No `true/false`. Instead, `false` is `0` or `""`
- String interpolation:

```
$team = "cats";
$yell = "Go $team"; #you need double quotes to activate interpolation. If you use single quotes, you just build the string as it's written.
```

- Control
    - `if, unless, while, until, for`
    - `unless` is the opposite of `if`, and `until` is the opposite of `while`
    - Except, we have a two forms.
    - Block form:

```
if (cond) { body }
elsif
else
```

    - Simple form:

```
statement if (cond);
```

    - `&&, ||, !`
    - `and, or, not`
    - You can combine `do { body }` with `until (cond)` for a ghetto `do`-loop
    - No `switch`
- Input from user
    - `$line = <STDIN>`: reads the next line, including the new line they enter
    - `<>`: This basically says read something from somewhere, which is the default STDIN location if you don't specify something else.
- Implied arguments and implied targets

```
<>; #puts the input in $_ since we don't specify location
print; #prints $_.
chomp; #gets rid of the new line in a string, in this case $_
```

- Lists or Arrays
    - Starts with `@`

```
@teams = ("str1", "str2", "str3");
@list2 = (5, 23.9, "bob");
$list2[1]; #returns 23.8
$myteam = $teams[1]
$teams = "packers"; #this is also legal. You can have 2 variables with the same name and different types
```

    - You can get the size of an array if you evaluate the array in a **scalar context**

```
$numteams = @teams;
$#teams; #returns the last legal index of teams
```

```
for($i=0; $i<@teams; i++) #this is bad
foreach $t (@teams) #do this instead
For (@teams) #this is also legal, and assigns each
element of @teams to $_
```

- Hash (also called an associative array)
  - Is like an array, but you can use anything as the index
  - Use `%` to denote a hash

```
$days{"Jan"} = 31;
$days{"Feb"} = 28; #we don't use % here because the
element at index Feb is a scalar, but days is a hash
$n = $days{"Mar"};
%days;
```

Homework Program #3: Suggests using a list for all values of the hand (because of Ace's multiple values)

March 19, 2018 (class notes)

"The secret of joy in work is contained in one word - excellence. To know how to do something well is o enjoy is"
-Pearl S. Buck

Perl continued
Arrays/lists continued

- Range operator
  - `@myList = 1..100; #(1,2,3,...,100)`
  - `@myList = "a"..."z";`
- Input from a command, or from a file
  - Need to make a **handle** like STDIN.
  - Use `open(handlename, source)`. For example: `open(IN, "command|")` or `$line = <IN>;`
  - Remember to close the command: `close(<IN>);`
- Common Perl paradigm on file or cmd
  - i/o: open or die:

    `open (...) or die "...";`

    Including $! On the end of die includes info on the error. Include \n to leave out some extra shit that idk?
- Slices and Swaps
  - Recall:

Scroll down!

```
@a = (10, 20, 30, 40, 50);
$[x] = $a[2]; #30
@y = @a[0, 2, 4]; #(10, 30, 50)
($v, $w, $x, $y, $z) = @a;
($v, $w) = ($w, $v); #swap the elements
@a[0, 2] = @a[2, 0];
```

- Helpful functions: `push, pop, shift, unshift`
  - `push/pop`: work on the right of the array
  - `shift/unshift`: work on the left end of the array
- Scalar vs list context
  - The meaning of expressions can depend on the context of use: scalar or list context

    Ex: `@a as above`

```
@b = @a; #list context. Copies the list
$b = @a; #scalar context. Number of elements in a
```

    Ex: Reverse function has different meaning in scalar and list contexts:

```
@test = ("ST", "AC");
@ans = reverse @test; #list context: @ans gets reversed
$ans = reverse @test; #scalar context. Glues together the elements of the list, and THEN it reverses the string of elements. Returns the reversed string. This is like overloading, but on the return value: the meaning depends on the context.
print reverse @test; #print only accepts lists. Anything given to it is promoted to a list for printing purposes.This line prints ACST
print "" . reverse @test; #dot expects scalars, so this prints CATS
```

Functions

- Defining functions
  -
```
sub functionname {
    body
}
```
  - But where are the parameters?! Don't need it, because every function takes a list. You can access the parameters by `@_`. It is **always passed by reference**. So `$_[0]++` will change the original element at position 0 in the function call.

    Ex:

Scroll down

```
&fun(2, 3, 7);
&fun(5);
#Use & on function calls for user-defined functions
sub fun{
     @_; #has only 1 element in the second call
}

sub max{ #two arg. Subroutine
     my ($a, $b) = @_; #This now makes local variables
that don't alter the function argument's original values.
     return $a if ($a > $b);
     return $b;
}

sub max { #many args
     my @args = @_;
     My ($i, $j, $k); #can use for indexing later in
function
     }
```

Scope

- By default, every variable is global! Be careful of this, especially when using indexing variables in loops
- So **always** use "my" to declare all local vars in a function.
- Perl can use either lexical or dynamic scoping.
    - When using `my`: lexical scoping

      Top level:

      ```
      my ($card, @dealerHand, @deck);
      ```

      Inside a sub:

      ```
      @dealerHand; #checks local scope first, and then checks
      the surrounding scope
      ```
    - Perl also supports dynamic scoping via `local`. **DON'T do this**

      Ex:

      ```
      local $v; #save current value of $v. Make a new var $v.
      At end of the scope, restore the old one.
      sub f{
           my $a = 401
           local $b = 50;
      ```

```
        &g();
    }

    sub g(){
        print "$a $b";
    }

    Main:
    $a = 20;
    $b = 200;
    &f();
    #prints 20 50
```

- Common in Perl programs:
  - `use strict; #helps debugging by making you declare variables before using them.`

Regular expressions

- Pattern matching operator: =~ and !~

```
string =~ pattern;
if ($str =~ /^\s*[yY]/) #does str begin with y or Y, ignoring all whitespace?
#\s matches any single whitespace character
```

Types

Type error: **Not** a syntax error!

```
x = expr;
```

Maybe expr only evaluates to one type, like a string. But suppose x has type double. Then, during semantic analysis, the semantic analyzer may be tasked with catching this error. This will come down to language philosophy

March 21, 2018 (class notes)

"Most of all I find coding in Go really really fun. This is a bad thing, since we all know that "real" programming is supposed to be grueling, painful exercise of fighting with the compiler and tools. So programming in Go is making me soft. One day I'll find myself in the octagon ring with a bunch of sweaty, muscular C++ programmers bare-knuckling it out to the death, and I know they're going to op the floor with me"

-Matt Welsh

Scroll down

Types

Type determines:

- Allowed values
- How variables are store and represented
- Allowed operations

"Type error" refers to when one type is expected, but another incompatible type is received.

Common terms:

- A language is **statically typed** if every variable has a declared type. (variables have types). Results in more compile-time checking
- **Dynamically typed** variables may not, but **values** have type. Results in more run-time checks. This is in a language like Perl.

In an object-oriented language:

- An object can have both a static type **and** a dynamic type. For example, in Java if we have

```
Integer w = new Integer(12); //static type of Integer
f(w);
f(Object obj){
     System.out.println(obj);
}
//At runtime the obj static type becomes Object. The dynamic type at runtime is Integer.
```

Common terms continued:

- A language is **strongly typed** if converting between unrelated types is not allowed, or requires a cast.
- A language is **weakly typed** if a variable can be converted to an unrelated type without casting. Some languages include Perl (int ←→ string) and C (malloc returns a void* and you can assign that to any pointer you want).

In C++, if you want to use a pointer of one type as a pointer to another type, you can use `reinterpret_cast`. Say `p` is an `int*`. You can then do `char* q = reinterpret_cast<char*>(p);`

You can think of the strong/weak classification as a sort of spectrum:

Strong → weak

Pascal, Ada | Java | C++ | C | Perl

Scroll down!

Type System

There are three aspects to a type system. It essentially determines what is or is not legal, and defines what things are type errors.

1) **Type equivalence** - Determines when two types are interchangeable. There are two general philosophies:
    a) **Name equivalenc**e - Each type name distinct. There is no nontrivial equivalence.
    b) **Structural Equivalence -** Two types are equivalent if they have the same underlying structure. For example, if you make a class with an age integer, and another with a temperature integer, they can be considered equivalent.

    Ex:
```
struct complex{
     double re;
     double im;
};
struct polar{
     double r;
     double theta;
};
struct complex z;
struct polar w;
z.re=1.0; z.im = 2.0; //1+2i
w = z;
//C uses name equivalence on structures, so this is not allowed in C.
```

    However, C uses structural equivalence on pointers.

    Another example:

```
int a[10];
int b[20];
void init(int c[], int n, int v){
     for(int i = 0;, i < n; i++){
          c[i] = v;
     }
}
init(a, 10, 2);
init(b, 20, 5);
//This is fine in C because you have structural equivalence. The size of the array is not part of the type. An array is just a pointer.
```

    In pascal, the size of the array is part of the type.

Legal in C: (scroll down)

```
int i = 4;
double* x = &i; //weird
int* p = &i;
int** q = p; //weird
```

2) **Type compatibility** - When an object of one type can be used where another is nominally expected, without casting.

   `double y = 15; //int literal, type mismatch. 15 → 15.0`

   Often, a **widening** conversion is ok, but a **narrowing** one is not.

   Widening: allowed values in destination include all possible values of source.

   Ex:

   `short → int → long in C/C++/Java`

   `float → double`

   `int → float or double //note that float has to keep track of the exponent in the last 7 bits, so you technically can lose some information of the int when storing the int in a float, since the float only has around 24 bits to store the base number.`

   Narrowing:

   `int r = sqrt(38.7);`

   C++, Java: Cast: `static_cast<int>(sqrt(38.7))`

   C++ also has 2 more cast types: `dynamic_cast` and `const_cast`

An implicit type cast is called a **coercion**.

```
double s = 4; //gets promoted to 4.0
Quaternion q(1, 2, 3, 4); //components turned to doubles
Quaternion r = q + 5.3; //is legal if you have a constructor that takes 1 argument and a definition for Quaternion + Quaternion. We coerced 5.3 → (5.3, 0, 0, 0)
```

The type compatibility spectrum is:

Stricter → looser

Ada | Java | C++ | C | Perl

3) **Type inference** - A language needs rules for inferring the type of an expression. For example:

   `int + double + float → double`

   `int / int → int (Java, C, …). In perl it returns a double`

<u>Types of types</u>

1) **Primitive** - int, float, double, char, bool (maybe), …. But even for something like chars, are we using ASCII or UTF16 encoding? Some more maybes: signed/unsigned, short/long. An odd one: BCD - binary coded decimal (COBOL), in which each digit is represented with 4 bits. Allows you to store fixed precision exactly. LISP has a rational type built in. Fortran has a complex type built in.
2) **Arrays** - Consider 2D arrays. A rectangle or jagged shape (number of elements in each row differs)? Are they row or column major? In Java, the 2D array is jagged and row-major. In C/C++, you can do either rectangular or jagged, but rectuangular is faster because it lets you get the element in 1 hit. They are also row-major. Fortran is column-major and rectangular. Pascal is rectangular. Most languages use row-major.

March 26, 2018 (class notes)

"The notion of expressions is a crucial unifying principle in Mathematica. It is the fact that every object in Mathematica has the same underlying structure that makes it possible for Mathematica to cover so many areas with a comparatively small number of basic operations"
-Stephen Wolfram

Types of types continued
Rectangular arrays in C:
`int a[10][20]; //10x20 rectangular array of ints. Stored row by row in memory because it's row-major`
So finding `a[i][j]` means `*(a + 20i + j)`. Note that you only go out into the memory once. But note that the 20 is important! The number of columns is part of the type! Ex:
`void f(int c[][20], int numRows) //`**`note that we can only call this function on arrays with 20 columns!!! The 20 is needed: for this function to find elements in c it needs the number of columns in the type. You can put something in the first [], and it will be ignored`**

But in a **jagged** array:
```
int nr = 10, nc = 20;
int **a = new int *[nr];
For (int i = 0l i < nr; i++){
     A[i] = new int[nr]; //initialize the arrays to be pointed to
}
```

There exists in memory an array of 10 pointers, and each element (each pointer) points to an array of 20 elements. So now, `a[i][j]` means `*(*(a+i)+j)`. Note that in this case, we are going

into the memory **twice**. However, now we are more flexible: routine taking `a` could work on **any** 2D array: `void f(int **c, int nr, int nc) {}`
You are free to make rows varying sizes if you want. In the example above, the shape of the resulting matrix is actually a rectangle. If we **do** make them varying sizes, we lose locality. Each row is in its own chunk of the heap, so it is slower to access elements because we don't know where in the heap the arrays are stored. If we knew where they were, we could just do some arithmetic to access any element based on the element we are at currently, since the layout is predictable.
Note that in C and C++, you need to first delete the rows, then the main array `(a)` when deleting a 2D array.

Read about these in the book!!
3) Associative arrays, or hashes, or dictionaries
4) Records and structures
5) Unions: There are different types of unions: **free** and **tagged**:

Free union:

```
union myUnion{
     int i;
     float f;
};
union myUnion u;
u.i = 4;
u.f = 3.7; //no check on what type is currently in u!
u.i = 4;
float x = u.f; //ok according to the compiler!
```

Tagged union:

This is a union, with an extra field for keeping track on what is in there now. Ada was one of the most strongly typed languages

6) Subtypes and subrange types:
Pascal:

```
type:
     digit - 0 ... 9 //a digit would then be an int restricted to
the values from 0 - 9
var:
     d: digit;
```

In Ada you can do subtypes in 2 different ways:

- Compatible with parent
- Not compatible with parent

7) Enumerations:

Ex: color can be `red`, `blue`, or `green`.

Example in C: scroll down

```
enum Color {red, blue, green};
enum Color x = green; //note there are no quotes on green. It's
not a string
enum Color y = blue;
//Is okay to be used in normal code, like in switches
int z = y; //in C this is fine, because the values stored in
red/blue/green in Color are just integers 0/1/2. Note you can
manually change what these integers are
y++; //y is now green.

enum Color {red, blue, green}
enum Apple {grannySmith, fuji, red} //the red gives a compile time
error because of reusing an enum value. So we change it to:
enum Apple {grannySmith, fuji, cortland}

Apple a = red; //Note this is legal in C! (type errors!). This
makes a grannySmith apple a.
```

C is very loose on enums! It's easy to make type errors.

Next: Basic enums in C++.

Then: Better enumerations:

C++ - 11 introduced an enum class.

Java has nice enumeration. (to be continued another day)

===New topic===

Logic Programming

Also known as rule-based programming or declarative programming.

In this language, a program is **a collection of facts/axioms together with some rules for making deductions and inferences.** You **don't** specify *how* a computation will work, contrary to an imperative language. Instead, you query the system, or **declare** what you want to know.

Hence, it is a declarative language. The system applies the rules to answer queries.

Ex:

Facts:

```
HenryIII is a parent of EdwardVI
```

```
JaneSeymour is a parent of EdwardVI
HenryVIII is male
JaneSeymour is female
HenryVII is a parent of HenryVIII (Scroll down)
```
Rules:
```
If x is a parent of y and x is female then x is mother of y
If x and y have the same parents then they are siblings
If x is a sibling of y and x is male then x is a brother of y
```

Queries:
```
Is AnneI a half-sibling of EdwardVI?
Who are the cousins of ElizabethI?
```

The system somehow applies the rules. It could try different possible values for variables. Would involve a good bit of backtracking. Read **16.1 - 16.4** for more information. The latter part of this chapter includes information on Prolog.
This is popular for AI, automatic theorem proving
It can be done in **Mathematica**

Rule-based approach to sorting a list: (this is a bad idea!)

- Rule that matches an unsorted list, and does something to improve it

March 28, 2018 (class notes)

It's the video of Obama saying not to do bubble sort.

Use FullSimplify for program 4!

Mathematica
S. Wolfram in the 1990s
Language for symbolic and numeric calculations
As a programming language, it supports imperative, functional, rule-based paradigms.
Interpreted
For math functions, it does exact computations if requested and if possible. If you give a decimal in your request, it spits out a decimal

Use = for immediate assignment
Use LHS = expr (evaluate expr and store it in LHS)
:= delayed evaluation

LHS := expr (wherever you see LHS later, evaluate expr and use that)
Use := for defining functions/rules
Use [] around function calls
Use () for regular parentheses in expressions
Use {} for lists
Use [[]] to get a list element (smallest index is 1)

Writing a function
You can't do f[x] := x^2 to define a square function in Mathematica, since Mathematica **matches patterns.** You need to do f[x_] := x^2

Include clear[d] at the top of your file

You can add more patterns: f[x_, y_] := (x+y)^3
You can have specific pattern rules, too!:
f[x_, x_] matches f(x, x) and f(t+5, t+5)
F[{x_, x_}] matches {5, 5}

When you have defined both f[x_, y_] and f[x_, x_], it will do the more specific one.

Ex: Rule-based factorial

```
fac[0] = 1
1
fac[n_] := n fac[n-1] //space is an implied multiply, btw
fac[5] //will return 120

//note that fac[3.5] will give an error. As will f[x]
//Need to give the program some notion of types to fix this
```

Types in Mma:
The "type" of an expression in Mma is its "head".
Primitive types/heads include:

- Integer
- Real
- Symbol
- Complex

Structured types:

- List → so {1,2,3} → List[1,2,3]
- Power

- Log
- Plus
- Times

In patterns, you can match on the head. You can do it with fac[n_Integer] to ensure the fac function only matches integers.

You also have the ability to match only when a condition on the params holds:

```
f[x_Integer, y_Real] := <RHS> /;  x+y < 10
```

So we can do:

```
fac[n_Integer] := <RHX> /; n > 0

:= Block[{local vars};
     Stmt1;
     Stmt2;
     Stmt3;
]
```

Example code for HW4:

```
Clear[d]
Clear[t]
D[Sin[t],t]
d[Sin[t_], t_Symbol] := Cos[t] d[u, t] //give it the rule for only
the derivative of sin
d[Cos[t_], t_Symbol] := -Sin[u] d[u, t]
d[Sin[5],5]
d[Sin[t+7], t+7]
d[Sin[w],w]
d[yy^2,yy]
Clear[x,y,t]
d[Sin[3t],t]
```

Derivative of Sin(__), with respect to t is actually cos(___) . d/dt(___)

```
d[Sin[u_], t_Symbol] := Cos[u] d[u, t]
d[Sin[3t],t] //returns Cos[3t] d[3t,t]
d[Sin[Sin[Sin[t]]],t] //returns Cos[t]Cos[Sin[t]] Cos[Sin[Sin[t]]]
```

Derivative of a constant is 0:

```
d[c_, t_Symbol] := 0 //but we need to say that c_ is a constant
somehow. So we do:
```

```
d[c_, t_Symbol] := 0 /;FreeQ[c,t]
```
More generally, you can say: `FreeQ[{a,b},t] //if a AND b are free of t`

Scroll down
Constant multiple:
Let's do Sin(3t)
You don't need a constant multiple rule if you use a product rule (?)
Product rule:

```
d[u_ v_, t_Symbol] := d[u, t] v + v d[v, t]
d[Sin[3 t], t]
d[t_, t_Symbol] := 1
```

y=u^v
Lny = vlnu
y'/y = v'lnu + v(u'/u)
Y' = y ( ) = (u^v)(v'lnu + v(u'/u)) = (u^v)(lnu)(v') + vu^(v-1)u'

April 4, 2018 (class notes)

"If you're masochistic enough to program in Ada, we're not going to stop you."
-Matt Welsh

P4 hints continued

```
Hanoi[]
Hanoi[] :=
        Block[{},
                Hanoi[];
                Hanoi[];
                Hanoi[];
        ] /; n > 1
```

Higher order derivatives
First get things working for just single derivatives
Just need something to start the recursive calls (derivative of a derivative)

```
D[x t Sin[x t], x, x]
D[x t Sin[x t], x, t]
```

Just need to match against any number of arguments, and then call the derivative again.

Matching multiple arguments:
x__ (double underscore matches 1 or more arguments)
x___ (triple underscore matches 0 or more arguments)

f[a_, b_, x___] := {a, b, {x}}
f[34, 23, 7, 2, 8, 1]
Returns {34, 23, {7, 2, 8, 1}}

Can use this grouping to do recursion fairly succinctly.

Hints on an integrator
Integrate[Cos[x], x] returns Sin[x]

Don't need to worry about returning +c term

int[Cos[t_], t_Symbol] = Sin[t]
int[Cos[w],] returns Sin[w]

int[Cos[t_ + c_],t_Symbol] := Sin[t+c] /; FreeQ[c, t]]
int[Cos[w+3],] returns Sin[w+3]

int[Cos[a_ t_ + c_], t_Symbol] := Sin[a t + c]/a /;FreeQ[{a, c}, t]
int[Cos[5w+z],] returns Sin[5w+z] / 5

Optional arguments
int[Cos[a_. t_ + c_.], t_Symbol] := Sin[a t + c] / a /;FreeQ[{a, c}, t]
The . says that the argument is optional. "Sensible initial value" ???. So the default for a_ becomes 1 and the default for c_ becomes 0.
You can also do:
a_:1 t_ + c_:0 to supply the default value
Note then that int[Cos[-w],w] still returns Sin[w]

Definite Integrals
Integrate[Cos[t], {t,0,Pi/2}] //Match against the expression, and a list. In the list, the first thing is the variable, and the next 2 things must be values, so they must be independent of the variable (in this case, t).
Be sure not to calculate the antiderivative twice.
In this case, we will need to use the Block structure and a way to substitute.
To substitute, we do:

x^2 + y^2 /. x -> 5 //Substitutes x with y
Multiple substitutions: /. {x->5, y->7}
Note that we need to handle any linear combination of the integrals:
cos(2t+1) and sin(5t-4) and e^(3t-1)
The linear combination would be something like:
4cos(2t+1) - 7sin(5t-4) + 6e^(3t-1)

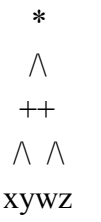
In this situation, we pull out the constant and evaluate the integral of each term, and then scale and sum

Now some non mathematica stuff:

Expressions (chapter 7), Control (chapter 8)
Expressions

- An expression is something that returns a value. (5+6, etc)
- In many languages, any expression is a legal statement
- Note that `x = 3` is an expression in C. It returns the value assigned
- Return value vs side effect.
    - A side effect refers to a change in the program state (a **change in some binding**)
    - `x+2*y` //has a return value, but no side effect because x and y don't change)
    - `Z = 14` //has a return value (in some languages) and a side effect
    - `delete p;` //has **no** return value, but has a side effect: releases the memory that p points to
    - `Cout << x` //has **no** return value, but has a side effect (printing)
    - In functional languages, you have far fewer side effects, and more reliance on recursion and return values.

Expression Syntax
Common:
Expression is (x+y)*(x+z)
AST is



- infix
    - Infix binary expression: op1 operator op2
    - An infix expression is the in-order traversal of the tree
- Prefix

        - * + x y + z w
        - Operator op1 op2
        - Is derived from a preorder traversal of the abstract syntax tree of the expression
- Postfix
    - Op1 op2 operator
    - Is derived from a post order AST traversal

Some languages use prefix or postfix! By why:
Postfix:

- Languages for device control
- Postscript (this is the name of a language)
- Forth
- "Reverse Polish notation"
- In postfix, you **never** need parentheses.
- Showed us his crazy ass old HP postfix calculator with no paren buttons

Prefix:

- Ambiguity when an operator has a unary and **binary context.**
- Eg: infix: -x, x - y //no ambiguity
- Eg: prefix: - x y //ambiguous
- The solution to the ambiguity is **Cambridge Prefix Notation**
- Infix: (x+y)*(w+z)
- Cambridge prefix gives: (* (+ x y) (+ w z))
- Always put function name or operator, then the arguments, and then put them in parentheses.
- This is essentially LISP and Scheme
- "Polish notation"

Precedence

- Important to look at the number of precedence levels when learning a new language!
    - C and Java have 15 levels
    - C++ has 18 levels
    - Perl has 24 levels (things like `&&` and `and`)
    - Pascal had 3
    - Ada had 6
    - APL had 1 xd

April 9, 2018 (class notes)

*"The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."*
-Edsger Dijkstra

Absolute value derivative:
Slope of tangent line to y = |x|: 1 x>0 and -1<0
Can get this result from Sign[x]
E ^ at + b (????????)

Expressions and Control (ch 7 and 8)
Associativity

- Left to right
  - +, *
- Right to left
  - ** (Exponentiation), = in C
  - In Ada, ** is nonassociative
- Nonassociative
  - Illegal to write a chain of things without specifying the parentheses.
  - < in Perl and Ada (they don't want to support if (a < b < c)). Note that line is legal in C and C++. Gives a syntax error in Java because it results in comparing Boolean to a number

Short-circuit evaluation
if (cond1 && cond2){ }
Evaluate cond1 first. If it is false, just stop and don't bother check cond2.
if (x != 0 && y/x < 2) { }
open file or die (In perl)

Order of Evaluation of arguments or operands
X + y
Can we count on x being evaluated before y? Would matter in the following situation: x+ ++x.
Also matters if you have f() + g() and the functions have side-effects that affect the other function.
Often times, you can't get it to evaluate in a consistent way.
In Java, it's left to right.
In C/C++, you can't count on a particular order of evaluation. This is because of optimization opportunities. If one of the terms was recently evaluated, you can quickly access it and use it.

Control
Control involves sequence, alternation, and loops/branching

- Sequencing
    - Sometimes you have a statement terminator, but sometimes you have a statement separator. For example, you don't need a comma after the last element in a list since it's a separator
- Alternation
    - If's earlier (ambiguity, end-if)
    - Switch-type statements have two ways to deal with it:
        - C/C++ way: manual break statements. Sometimes you omit the break for two cases to be handled identically:

```
Case "d":
Case "D":
        Stuff;
        break;
```

        - In some languages, it is essentially an automatic break (like an if then else)

```
Case x of
        3: <statement>
        4 .. 10: <statement>
        11, 13: <statement>
        Else <statement>
End;
```

        If you have a case statement but it does not match any of the cases, it gives a runtime error in Pascal. Gives a compile time error in Ada. Gives no error in C. C just will do nothing
- Loops
    - Many kinds of loops:
        - Logically controlled
            - While, until, do while
        - Enumeration-controlled
            - For loop
        - Structural
            - Foreach in an array
            - For (x:array) {}
- Branching
    - Break, return, goto, continue
    - Goto works in C. You can label a statement and then use goto myLabel
    - Perl has last, next, redo

New topic:
<u>Object Oriented Languages</u>

Inheritance:

- Making a derived class (extends in java)
- Overriding a method
- Interface
- Abstract base class

In OOP, **the program is a collection of classes.** When you run a program, you create objects (instances of classes) and you invoke methods on those objects. The objects interact with each other. It is a data-centric way to program.

Defining a class in Java:
Access is specified per item (public, protected, private). Also there is package, but we don't care about that one.
Public means anyone can see it (objects of other classes can see them)
Private means only methods of this class can see them. Note this is class specific, not object specific. So 2 objects of the same kind can call private methods of each other
Protected means it can be seen by methods of this class or classes derived from this class.

```
public class ClassName {
        public ClassName()
        Public int methodOne()
        private int x, y;
        private int helper()
}
```

In C++:
Meanings are exactly the same as java, but they are syntactically different.
Default is private in C++

```
class ClassName{
        Public:

        Protected:

        Private:
}
```

Extending:

```
public class ClassName extends BaseClass {
```

```
        public ClassName()
        Public int methodOne()
        private int x, y;
        private int helper()
}

class ClassName: public BaseClass{
        Public:

        Protected:

        Private:
}
```

Note that you can change public BaseClass to be either protected or private, and it means something different.

April 11, 2018 (class notes)

"We have now disarmed the kangaroos and it is safe again to fly in Australia"
-Anna-Marie Grisogono

Inheritance in C++/Java continued

Uses for Inheritance

- Extension
    - Derived class adds additional fields and methods relative to the base class
- Specialization
    - Derived class needs to alter some behavior of the base in some way - overriding methods. Eg: Special algorithm for the case of the derived class.
- Specification
    - Base class left some cases open
    - Base class mandates some interface, but does not handle implementation for everything
    - Need a derived class just to have a class you can make objects with
    - (This is what's happening in P5): Gives abstract base class for a player and shedgame. Shedgame is easy, it's just denoting which cards have special triggers. Player is the hard part.

Main ideas in (public) inheritance

Suppose we have a base class B, and a derived class D.

Java:

```
D d = new D(...);
f(d) //where void f(B b)
B b = d;
```

A derived object can be used anywhere a base object is expected.

So, we now have two types of objects in C++:

- Static (static type of `b` is `B`)
- Dynamic. At runtime, this is what kind of object it really refers to. When you call f, the dynamic type of `b` could be either `B` or `D`.
    - That rule is called the **substitution principle**. It is often shortened as a "is a". Every instance of a derived class is an instance of the parent class.

Storage:

Suppose we have a D object. It looks like:

```
_________
|<B fields>|
|---------- |
|<D fields>|
|_________|
```

Constructing Derived objects

We can't assume that the D constructor can initialize the B fields (ie if the B fields are private). The derived class constructor must rely on the base class constructor. It must have a way to invoke a constructor of its base class.

In **Java**, you issue `super(args)` as the first line of the derived class constructor.

In **C++**, call the base class constructor explicitly: `(B())`. But you don't do it in the body of the function! You call it in the **initialization list** (of the derived class).

Init lists in C++

For constructors but not necessarily with inheritance.

```
MyClass::MyClass(int n){
        //Field called num
        num = n;
}
```

Equivalently, you can do:

```
MyClass::MyClass(int n) : num(n){
}
```

That thing after MyClass is the init list of field values. If one of your fields is a constructor, you need to do this! So you initialize up there, with the correct params

Why use an init list?

- More efficient for objects
- Needed in inheritance: This is where you call the base class constructor
  D::D(..) : B(..), num(n){ } //**Put the base class constructor first!!!**
- If the field is a const, or a reference, you **must** use init list.

If the fied was `const int num;` then in your constructor you have to use num(n) in init list.
Similarly, int #

Hiding

What if `D` and `B` both have a field with same name? Eg: `int x`
That's legal in C++. `D`'s `x` **hides** `B`'s `x`. If you use "`x`" in `D`'s method, it refers to `D`'s `x`, not `B`'s.
To access `B`'s `x`:
Java: `super.x` (super refers to the current object (this) as member of base class)
C++: `B::x`

What if `B` and `D` both have **methods** with the same name?
Java:

- If the `B` method and `D` method have different signatures, all is ok.
- If they have the same signature **and** return type **and** access level in the derived >= base, all is ok!
  - D overrides the function of B

  Ex:

```
B: public void f(int)
D: public void f(int)
D d = new D(...);
myfunc(d);
d.f(3); //calls D's f(int)

void myfunc(B b){
     b.f(2); //At runtime in the cal above, b refers to a D
object The dynamic type is D. So it calls D's f(int)!!
}
```

  This is an example of **Dynamic dispatch**: Runtime decision on what function is actually called. Depends on dynamic type of `b`
- Other cases: Eg: same signature but different return type, or maybe the same signature but different access level. These are errors at compile time.

C++:

Note that in C++, the signature is the sequence of parameter types, plus the `const` method or not

- You can hide methods in C++

  Both public:

  ```
  B: int f(int)
  D: int f(int, int)

  D D(..);
  d.f(3, 4); //ok
  d.f(2); //error as is
  d.B::f(2); //ok
  //Or you can turn on dynamic dispatch for B's f(int) with
  some special syntax. Note this is usually a weird case, and
  normally you just override the funciotn
  ```

- Same signature, return type, and access level.
  - Probably want D's method to override B's. So ask for it with the keyword "`virtual`"

    ```
    B: virtual void f(int) //enables dynamic dispatch
    //Later: in b.f(3) or d.f(2): dynamic dispatch
    ```

    Note this flips the default relative to Java. In Java, making a method `final` turns this off (no subclasses can override the method)

  Each class with at least one virtual method has a "virtual function table". This table holds a list of all the functions that the class will be using.

To get overriding in C++:

- Same signature, return type, access ok (as in Java)
- Base class method is `virtual`. This is a flipped default with respect to Java
- Invoke the method via reference of pointer (automatic in Java).

```
D d(..);
go1(d);
go2(d);
//Both D and B have void f(int);
//B's is virtual

Void go1(B& b){
     b.f(2); //gives us the dynamic dispatch.
     //b in here is a D object, with both B and D stuff
```

```
     //A reference of type B (via B&) can refer to derived classes
of B
}

Void go2(b b){
     b.f(3);
     //Causes issues because b is just a B object. When we call
it, we only get the B stuff. This causes a slice. This is the
"slicing problem" in C++: assigning a derived class object to a
base class type (no reference or pointer). If you pass by
reference, it's okay
}
```

April 16, 2018 (Class notes)

"For those who have wondered: II Dont' think oop is a structuring paradigm that meets my standards of elegance"
-Edsger Dijkstra

Inheritance in Java and C++ continued

Recall that to override in C++ you need to have:

- The same signature and return type
- Base class must be declared `virtual`
- Have to call the method from a reference of pointer of the object (which you always do anyways, in C++)

Note: If any methods are `virtual`, it's good habit to make the destructor `virtual` too. Else, you may try to call a base class destructor on a derived class. This can lead to unpredictable things, like crashing or memory leaks. It's a really bad bug to have!

Abstract methods and classes

Java

- Can declare a class to be abstract if you want to.
  `Public abstract myclass`
  Means you **CAN'T** make object of this abstract class
- A java class can't have abstract methods
  `Public abstract int f(int a)`
- An abstract method has **no body**.

- It's just interface - no implementation
- A class with at least one abstract method **must** be declared abstract
- If a class extends an abstract class, the derived class must override the abstract methods and provide implementation in order to be able to make objects of the class. (Else, D [derived class] would need to be abstract too)

C++

- No keyword or syntax for "abstract" class.
- Same rules though for abstract methods, and just a different syntax.
- Syntax for an abstract method:

```
virtual int f(int) = 0; //the 0 places a null pointer in the virtual function table for this class
```

- The C++ term for this kind of function is a **pure virtual function**

Final

- Java has the "final" keyword. A final **class** cannot be extended, and a final **method** cannot be overridden

Program 5 (hw)

Turn on C++11 on this hw!

C++ Standard Template Library

- Container classes
  - Lists, sets, vectors
- Some algorithms for sorting/shuffling/searching etc

Ex: To use stacks, we do `#include <stack>` and then do `stack<int> s;` to make the stack of integers s.

```
#include <string>
#include <stack>
stack<int> s;
stack<string> t;
s.push(5);
s.push(11);
Int w = s.top(); //peek. Returns 11
s.pop(); //does not return!
s.empty();
```

Vectors

Probably want to use a vector for the hand of cards, so you can get into the inside of it. It is like an array class in Java. It's a smart version of a C++ array.

```
#include<vector>
vector<int> v;
v.push_back(10); //puts 10 on the end of the array
v.push_back(15);
v.push_back(20);
int x = v[2]; //20
v.size();
Int z = v.pop_back(); //returns and removes the last element
Random_shuffle(v.begin(), v.end()); //begin() returns a "smart"
pointer. Have to do #include<algorithm>
```

3 things about P5

1) Write three concrete extensions of ShedGame

```
Class Crazy8 : public ShedGame{
Public:
Crazy8();
Bool isWild(const Card&) const;
...
}
Crazy8::Crazy8():ShedGame("Crazy Eights"){}; //initialization
Bool Crazy8::isWild(const Card& c) const{
     //just test if c is an 8 so you look at the value of
c.getRank() and compare it to the ENUMERATION value, NOT the
integer 8
     c.getRank()==Card::eight;
}

//And then you do something similar to set the hand size for each
game
```

2) Finish ShedGame. It is currently missing **2 methods**: nextPlayer() and restock()

Restock()

- Save the top card in the discard stack, and put it to the side.
- Return other discards to the stock
- Shuffle the stock
- Put the saved card back on the discarded stack

What if there's only one card in the stock, and all other cards are in the players' hands? In that case, the above algorithm wouldn't work. So we would have to get a new deck if the discard

stack in step 2 is `none`. To prevent infinite loops, you should peek at the card that the player picks, to see if they got a wildcard, in which case you have to prevent them from picking more cards.
Scroll down
nextPlayer()
Is called when the player is done with their turn and the dealter wants to know who they're going to ask next. Needs to update `curr` (index into the player vector for who the current player is). Often, you just increment curr and wrap it around the vector if needed. **But** there is a card that can reverse the order of play! So in this case, you would want to subtract one, instead of add one. And what if someone played a skip? The game needs to know about these plays. There is a variable `incr` to keep track of this. If someone plays a skip, then `incr` is one more (so instead of adding 1, you add 2). So you just add `incr` to `curr` and wrpa when needed. If `incr` wasn't +/- 1, you need to do the increment and then reset `incr` to +/- 1.

```
curId = player[cur] → getId(); //keeps track of some shit internally. Something to do with the order of players for each game. This asks the player what their id number is. Don't know why it exists. He said he would add it to the code because it's not obvious
```

3) Write the player.

April 18, 2018 (class notes)

"In computing, elegance is not a dispensable luxury but a quality that decides between success and failure."
-Edsger Wijkstra

P4 remarks

1) Keep the differentiation simple:
   d[Sin[u_], t_Symbol] := Cos[u]d[u,t]
   D[u_ +v_, t_Symbol] := …
   D[c_ u_, t_Symbol] := cd[u,t] /;FreeQ[c,t]
2) Higher derivatives
   d[u_,t_Symbol, rest__Symbol] := d[d[u,t], rest]
3) Integrals:
   int[Sinh[a_. t_ + b_.], t_Symbol] := Cosh[at+b]/a /;FreeQ[{a,b},t]
   Need the t in Sinh to be the same as the t_Symbol.

Back to notes now… scroll down
Other kinds of inheritance in C++
Before, we used:

```
Class D : public/private/protected B{ .. }
```
The access level determines how visible the public base class components are in D:

a) **Public**: public components of B are effectively transported to public in D. i.e. clients of D can call B's public methods.
```
D d;
d.basemethod(); //ok!
```
This is the same thing as Java `extends`.
Substitution principle holds (D is a B)
D inherits B's interface

b) **Private**: public components of B are effectively in the **private** zone of D. so clients of D cannot access B methods, but **methods** of D **can** access B methods. No substitution principle. You can't use D object where a B is expected.
```
D d();
f(d)
----
Void (B &b)
     Illegal
```
Instead, you can think of it as **D is implemented in terms of B** or **D has a B inside it**.
This is similar to D declaring an object of class B in its private section.

Private inheritance from list	Field list (composition: one object inside another)
(inside a D method): `add(item)`	`mylist.add(item)`

This is called inheritance of implementation. One advantage of this is private inheritance model is that you can access protected members of B. Under composition: public only.

c) Protected: you put the protected stuff from B and put it in D. ??????? It's in the textbook.

<u>Multiple inheritance</u>

Suppose we have a class for representing musical instruments in an orchestra.
Notation: derived → base
{violin, cello, piano} → String instrument
{piano, timpani} → Percussion instrument
{} → Brass instrument
{} → Woodwind instrument

In Java, `public Piano extends StringInst, PercussionInst{...} //Gives an error in java!`

In C++: `class Piano: public StringInst, publicPercussionInst{ …}`
`//Fine in C++. wasn't allowed in the very first versions of C++. Note that you need to say public TWICE. You can publicly inherit from 1 class and privately from another!`

Java solution:
Limited multiple inheritance. Use interfaces as the solution. A java interface is essentially a purely abstract class. New keyword: `interface` in place of `class.` Every method is public and abstract. There is no per-object data: any field must be `public static,` and `final.` Note these are all automatic. For example: `public interface Comparable{ int CompareTo(Object obj); }`

`Public class D extends B implements Comparable{ … }`
Here: D implements something with an abstract method, so we must provide a definition for ComparTo in order to be concrete
Substitution rule: works on D for both B and Comparable. Ex: could make a method sort taking in a list of Comparable's
`Public class D extends B implements Comparable, Cloneable { … }`
In java, you can only extend one class, but you can implement as many interfaces as you want. Interfaces are extremely restricted, though.

C++ Solution:
Full multiple inheritance is okay!
You need to say the access level on each one
```
Class D:public A, public B {.. }
D d;
```
D constructor must call both A() and B() in init list.
d looks like:

| A stuff |

| B stuff |

| D stuff |

Creates some problems though: scorrl

1) **Ambiguity**

Classes	Methods
`ArmedCharacter`	`draw()`
`CardPlayer`	`draw()`
`DisplayedObject`	`draw()`

```
class Gunslinger:public ArmedCharacter, Public CardPlayer, public
DisplayedObject{ .. }
Gunslinger blackhat;
blackhat.draw();
```

So what does .draw() do? It results in a compile time error

Some fixes are:

- Kludgy fix: blackhat.CardPlayer::draw();
    - But what if Gunslinger wants to **override** a draw()?
- More complicated fix: Have to introduce intermediate classes to effectively rename draw() functions

2) Redundancy

Back to instrument example: Piano is both a string and percussion instrument. But we also then add higher class, called Instrument. So both string and percussion are now derived from instrument:

Instrument
^ ^
| |
String percussion
^ ^
| |
Piano

Makes the diamond of inheritance that we don't like.

A
^ ^
| |
B C
^ ^
| |
D

D makes B, C

B makes A

C makes A

D d;
So our d object has 2 of the A stuff in it:

_____
| B stuff |

| A stuff |

_____
| C stuff |

| A stuff |

_____
| D stuff|

_____

Suppose int a in A is protected.
And then in a D method we say: a = 5;
Which a are we referring to? It's ambiguous and scope resolution doesn't help.
The fix is **virtual inheritance**. If you will have a diamond/cycle in your inheritance tree, then you should inherit virtually from the top (repeated) class.

```
Class B : virtual public A {...}
Class C : virtual public A { …}
```

This means that the construction of A part is no longer the responsibility of the immediately derived class. It is now the responsibility of the **most** derived class (in this case, D. The class farthest from A).
Now:
D d; The d constructor makes A, B, **and** C. The d object looks like:

_____
| B stuff |

| pointer to A stuff |

_____
| C stuff |

| pointer to A stuff |

_____
| D stuff | **(scroll down!)**

| A stuff |

_____

Now we have no redundancy!
Remember that in Java interfaces there is **no** per-object data, so this kind of A stuff data can't be repeated.

A common use of multiple inheritance in C++:
Publicly inherit from B1 (interface which you actually care about/want to implement)
Privately inherit from B2 (data structure giving you some storage space)

3) **f**

April 23, 2018 (class notes)

"I don't like Pascal. It's a straitjacket. ...Pascal has stopped me from writing good programs"
-Bjarne Stroustrup

Reach ch9 and 15.1-3 (?)
Presentations evaluated partially on:

- Practical aspects (history/design goals/applications/relationship to other languages in course)
- Technical aspects (type system/scoping/expressions/control structures/unusual features)

1=poor, 2=uneven but more negative than positive, 3=uneven, but more positive than negative, 4=very good, 5=excellent

gFunctions

1) Terminology

Similar terms:

- Function - returns a value
- Method - OOP
- Procedure - Don't return a value
- Subroutine -

Eg: Pascal has procedures and functions. C only has functions but introduced void type to have blocks that return nothing.

Input variables for a function: formal parameters.
The values supplied at the function call are the (actual) arguments.
Ie: void f(int x) (this has a parameter)
f(3) (this has an argument)

2) Function calls

Need some way to match the arguments to the parameters. The normal way of doing this is positional:

Int f(int x, int y) …

f(3, 4) //The position of 3 matches the position of x

Alternatively, you could do it in a named fashion:

f(y=>4, x=>3) (ada does this)

Or, default arguments:

F takes an int x with default value of 4

Then in a call, you can do f(17) and even f().

To do this, you can write: void f(int x = 4) //works in C and C++

Application in C++:

```
Rational()
Rational(int)
Rational(int, int)
```

Can unify the first 2 into: `Rational(int = 0)`

Sometimes you can have a functions with a variable number of arguments supported (like Perl: no formal param list, you just use @_). C and C++ allow this as well (because of printf). You can define a function with variable number of arguments in C as `void printf(const char*, …)`. The formal name is **variadic** functions. This is not really used anymore, though.

3) Return values

   Common: `return expr;` //Note that this can only return **one** item

   Some languages: allow returning multiple items. `A,b,c = f(x,y)`. This works in Lua (and Ruby, but fuck ruby xd)!

   In Pascal, though: Pascal has functions, but there is **no** return statement. To specify a return value, you assign the value to the function name. The function name is considered write-only: you can't check the value stored in the function name.

4) Passing values to parameters

   "How are arguments and parameters linked?" There are 5 different ways to pass values:

   a) Pass by value (PBV) - evaluate the argument, and assign it to the corresponding parameter. The parameter behaves just like a local variable. So changing the parameter has no effect on the argument. This is sometimes referred to as "**copy in**". In C, everything is passed by value.

   b) Pass by reference (PBR) - Parameter is a reference to the corresponding argument. If you change the parameter, you change the argument. Often, the argument must be an L-value. PL/I would do pass by reference on any L-value. You can force it to do pass by value by changing the argument: f(x) becomes f(x+0) or f((x)). This might have been forced way of interpreting parameters to save memory. Perl always wants to do pass by reference.

c) Pass by value-result - Exactly like PBV on the function call, but at the **end** of the function, the value of the parameter is **copied back** to the corresponding argument, which has to be an L-value. Copy-in, then copy-out.

```
Procedure f(x, y:in out Integer)
Begin
        x:=x+1;
        y:=y+1;
End
a:=8;
b:=9;
f(a,b);
Print a,b; //9, 10

c:=20;
f(c,c);
Print c; //21 (pass by reference would mean it returns 22)
```

d) Pass by result - Out parameter. It's just copy-out semantics. Nothing copied at start. Value of parameter is copied to the corresponding argument at the end of the procedure.

```
g(x:in Integer, r:out Integer)
g(3, ans)
```

Method for getting multiple return values

e) Pass by name - Algol60 used this. It's similar to a macro expansion. Meaning: when the function is called, substitute each occurrence of the formal parameter by the actual argument. Name can be any L-value. Similar to pass by reference, but it's **not** the same.

```
Call: sum(i, v[i], 1, 1000) //first two are PBN, last 2 are PBV
Int sum(k, ak, start, stop){
        Int s = 0;
        For k from start to stop do:
                S += ak
        od
        Return s;
}
```

Sums the elements of the array

5) Coroutines

Lua, Simula.

This of two or more routines as being active at once. One coroutine can hand off control to the other, back and forth. Where you continue in the coroutine is where you last left off (once you regain control).

April 24, 2018

"Some people prefer not to commingle the functional, lambda-calculus part of a language with the pats that do side effects. It seems they believe in the separation of church and state"
-Guy Steele

Functional programming
Historical background

- In development of CS (Theory of computation), there were three main schools of thought:
    - Predicate calculus, propositional logic. Things you can compute are things you can logically deduce. Dates back to around the 1890s. **Gave us rule-based programming.**
    - Turing machines (in the 1930s). The turing machine is a FSM with memory. This is basically **imperative languages**. Needs states (variable settings).
    - Lambda calculus. Alonzo Church, from the 1930s. Lots of nested function calls. Lots of nested function calls. A computation is like a bunch of nested compositions. This is **functional programming**. You have very little amount of state, so not much of things like for loops. The earliest of these is LISP. Some others are ML, Haskell, and Mathematica, and APL.

Requirements for functional programming languages

1) First-class functions - You can use a function just like you can use an ordinary variable. Most notably, you can pass a function to a function! You can store a function in a data structure. You can return a function from a function. In C/C++, functions are **not** really first-class, but you **can** pass a pointer to a function to a function (give the function an address to another function). Eg:

```
Double (*pf)(double);
//pf is a pointer to a function that takes 1 double and returns a double.
Pf = &sqrt;
(*pf)(5.0);
```

   This could be handy for something like a numerical library.
2) Anonymous functions - The ability to define a function without a name. Church used "lambda" as his version of a nameless function. Think python! The Church notation was

something like: λx: x*x. In Maple: map(x→ x^2, list); Mathematica supports the same idea: Map[#^2&, list]

3) Powerful facilities for lists - Since there are no for loops, we use recursion on lists very often
4) Garbage collector
5) Structured return values (list)

Pure functional programming

No state! So, variable values don't change. No mutable variables. No side effect; all is done via return value. Recursion, not iteration. Produces "**referential transparency**": A function called on a particular input always behaves the same way. There is no dependence on state.

Often, a functional language will have a way to set state. It might be awkward or not often used. In Lisp: (setq …)

Mathematica code for finding the primes from low to high that equal 1 when divided by 4

```
Select[Range[low, high], Mod[#, 4] == 1 && PrimeQ[#]&] //Length
```

C function for computing sum of f(i) from a to b (sigma shit on a computer LUL)

```
Int sum (int a, int b){
     Int ans = 0;
     For (int i = a; i <= b; i++){
          Ans += f(i);
     }
     Return ans;
}
```

In mathematica:

```
sum[a_Integer, b_Integer, f_Function]:= If[a>b,0,[a]+sum[a+1,b,f]]
```

In Lisp:

```
(defun (sum (lambda (f a b))
     (if (> a b)
          0
          (+ (f a) (sum f (+ a 1) b))
     )
))
```

Scroll down

Recursion vs Iteration

- Seems like iteration is better for practical reasons: less overhead. Recursion can overflow the stack quite easily)
- The solution to the overflow mentioned above: If possible, use **tail recursion**. Here, the recursive call is the **last** thing in the recursive body. This way, you don't have to save anything on the program stack. It basically says start over again, with these new values as the parameters.

Eg:

Non-tail recursive factorial n!

```
Int fact(int n){
     If (n == 0) return 1;
     Return n*fact(n-1); //not tail recursive because the last
thing you do is the multiplication by n
}
```

Tail recursive factorial n:

```
Int fact(int n){
     If (n == 0) return 1;
     Return auxFact(n, 1);
}

Int auxFact(int v, int t){ //Invariant: n! = v! * t
     If (v == 1) return 1;
     Return auxFact(v-1, v*t);
}
```

April 30, 2018 (class notes)

"Lemme expand: I made a prototype, the my employer threw millions of dollars at it"
-Graydon Hoare

Things to talk about:
Referencing with &
Arrays fixed size?
Parameters: how are they passed?
How to delcaer diff structures like array
Compare to other languages (can do it as we go. Ie bring up some similarities of a feature to some other language as we introduce it)

Rust: Aidan and Grahame
Thread safe. No seg faults.
Compiled. Multi paradigm.
No garbage collector.
Ownership/borrowing: every value is owned by a variable. Can only have 1 owner at a time. When out of scope, value is dropped. Shared with the reference &.
Weak word: language wants to you not use it, but you can if you want

LISP: Jack, Antoinette, and Adam
Multi paradigm
ARPA
AI research
Goals: commonality and portability.
List vs non-list (Atom) types
Strongly typed
Dynamically typed

Go
Concurrency!
Goto statement
Kinda both dynamic and static typing

Javascript
1995
6 primitives
33 key words, I think.
Arrays are more similar to a hash. Elements are not stored in memory contiguous.
Primitives are immutable.
Pass by value for primitives
Pass by reference for all else

Ada
First internationally standardised object oriented language
Types are incompatible (can't do 1 + 2.3)