ECS 140A Discussion 1

Ty Feng, Chris Fernandez

2:10-3:50pm Thursday, Aug 10 2023

UCDAVIS

Today’s agenda

Introductions (10 min)

Overview of different languages (5 min)

Review of preliminary concepts (15 min)

Introductory lecture material (20 min)

Syntax, derivations, parse trees (30 min)

Java tutorial (15 min)

1. Introductions

Ty Feng

2nd year MS student in Computer Science

B.S. Computer Science from University of Miami, FL

Software engineer in industry for 3 years (coding mostly in Java, TypeScript, and Python)

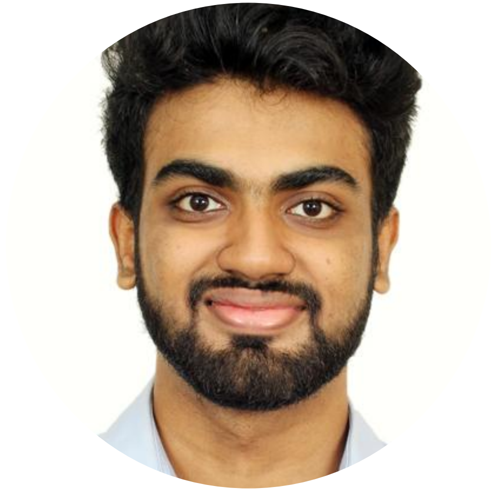
Research interests are AI/ML/CV

When I’m not coding or doing research, I play bass!

Office hours: 8-10pm Mondays and Thursdays on Zoom

tyfeng.com | tyfeng@ucdavis.edu | *Zoom:* [*https://ucdavis.zoom.us/j/3199978352?pwd=bTFISEpRV01jelcxTHlVVmUweS9PQT09*](https://ucdavis.zoom.us/j/3199978352?pwd=bTFISEpRV01jelcxTHlVVmUweS9PQT09)

1. Introductions



Chris Fernandez

2nd year MS student in Computer Science

B.E Computer Engineering from University of Mumbai, India

Research interests - ML, AI, HCI

I enjoy playing sports, especially soccer in my free time

Office hours: 7-9pm Tuesdays and Wednesdays on Zoom

cjfernandez@ucdavis.edu | Zoom:
https://ucdavis.zoom.us/j/94187714024?pwd=Q2FPekFVRW1acE4veXBJbEZCTWpPZz09

Student introductions

(in groups of 3-4 for 5 minutes)

Name, Major, Year…

What programming languages do you know?

What do you hope to get out of this course?

Favorite dev environment/tools?

Career goals? …

2. Just for fun: Same program, different languages

Let’s talk about programming languages!

* Participation, active engagement, and discussion strongly encouraged ... each of us may be expert in other languages and learn from one another

2. Just for fun: Same program, different languages

Rosettacode:

https://www.rosettacode.org/wiki/99_Bottles_of_Beer

C

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        if(argc == 99)
                return 99;
        if(argv[0] != NULL){
                argv [0] = NULL;
                argc = 0;
        }
        argc = main(argc + 1, argv);
        printf("%d bottle%c of beer on the wall\n" , argc, argc == 1?'\0':
's');
        printf("%d bottle%c of beer\n", argc, argc == 1?'\0': 's');
        printf("Take one down, pass it around\n" );
        printf("%d bottle%c of beer on the wall\n\n" , argc - 1, (argc - 1) ==
1?'\0': 's');
        return argc - 1;
```

Python

```
i = 100

while i > 1:
    i -= 1
    print(f"{str(i)} Bottles of beer on the wall\n{str(i)}
bottles of beer\nTake one down, pass it around\n{str(i -
1)} Bottles of beer on the wall.\n")
```

OR in one line:

```
for i in range(99,1,-1): print(f"{str(i)} Bottles of beer
on the wall\n{str(i)} bottles of beer\nTake one down, pass
it around\n{str(i - 1)} Bottles of beer on the wall.\n")
```

Java

```java
import java.text.MessageFormat;

public class Beer {
    static String bottles(int n) {
        return MessageFormat.format("{0,choice,0#No more bottles|1#One bottle|2#{0}
bottles} of beer", n);
    }

    public static void main(String[] args) {
        String bottles = bottles(99);
        for (int n = 99; n > 0; ) {
            System.out.println(bottles + " on the wall");
            System.out.println(bottles);
            System.out.println("Take one down, pass it around");
            bottles = bottles(--n);
            System.out.println(bottles + " on the wall");
            System.out.println();
        }
    }
}
```

Swift

```
for i in (1...99).reversed() {
  print("\(i) bottles of beer on the wall, \(i) bottles
of beer.")
  let next = i == 1 ? "no" : (i-1).description
  print("Take one down and pass it around, \(next)
bottles of beer on the wall.")
}
```

Rust

```
fn main() {
    for n in (0..100).rev() {
        match n {
            0 => {
                println!("No more bottles of beer on the wall, no more bottles of 
beer.");
                println!("Go to the store and buy some more, 99 bottles of beer on 
the wall.");
            },
            1 => {
                println!("1 bottle of beer on the wall, 1 bottle of beer.");
                println!("Take one down and pass it around, no more bottles of beer 
on the wall.\n");
            },
            _ => {
                println!("{0:?} bottles of beer on the wall, {0:?} bottles of beer.", 
n);
                println!("Take one down and pass it around, {} bottles of beer on the 
wall.\n", n-1);
            },
        }
    }
}
```

APL

```
bob  ←  { (⍕⍵), ' bottle', (1=⍵)↓'s of beer'}
bobw ←  {(bob ⍵) , ' on the wall'}
beer ←  { (bobw ⍵) , ', ', (bob ⍵) , '; take one
down and pass it around, ', bobw ⍵-1}
↑beer¨ ⌽(1-⎕IO)+⍳99
```

Brainfuck

```
>+++++++++[<+++++++++++>-]<[>[-]>[-]<<[>+>+<<-]>>[<<+>>-]>>>
[-]<<<++++++++++<[>>>+<<[>+>[-]<<-]>[<+>-]>[<<+++++++++>>>+<
-]<<-<-]+++++++++>[<->-]>>+>[<[-]<<+>>>-]>[-]+<<[>+>-<<-]<<<
[>>+>+<<<-]>>>[<<<+>>>-]>[<+>-]<<-[>[-]<[-]]>>+<[>[-]<-]<+++
+++++[<++++++<+++++++>>-]>>>[>+>+<<-]>>[<<+>>-]<[<<<<<.>>>>>-
]<<<<<<.>>[-]>[-]++++[<+++++++++>-]<.>++++[<+++++++++>-]<++.>+
++++[<++++++++++>-]<.><++++++..---------.--------.>>[>>+>+<<<-]>
>>[<<<+>>>-]<[<<<<++++++++++++++.>>>>-]<<<<[-]>++++[<++++++++
+>-]<.>++++++++++[<++++++++++>-]<--.----------.>+++++++[<------
---->-]<.>++++++[<+++++++++++++>-]<.+++..+++++++++++++.>++++++
++[<---------->-]<--.>+++++++++[<++++++++++>-]<--.-.>++++++++
[<---------->-]<++.>++++++++[<+++++++++++>-]<++++.-----------
-.---.>+++++++[<---------->-]<+.>+++++++++[<++++++++++++>-]<-.
>++[<------------>-]<.++++++++++++..>++++++++++[<---------->-]<
-----.---.>>>[>+>+<<-]>>[<<+>>-]<[<<<<<.>>>>>-]<<<<<<.>>>+++
+[<+++++++>-]<--.>++++[<+++++++++>-]<++.>+++++[<++++++++++>-]<.
><+++++..---------.--------.>>[>>+>+<<<-]>>>[<<<+>>>-]<[<<<<++
++++++++++++.>>>>-]<<<<[-]>++++[<+++++++++>-]<.>++++++++++[<++
+++++++>-]<--.----------.>+++++++[<---------->-]<.>+++++++[<++
+++++++++>-]<.+++..++++++++++++++.>++++++++++[<---------->-]<
-.---.>+++++++[<+++++++++++>-]<++++.+++++++++++++.++++++++++.
------.>+++++++[<---------->-]<+.>+++++++++[<+++++++++++>-]<-.
```

```
-.----------.>+++++++[<---------->-]<+.>+++++++[<++++++++++>-
]<--.++++++++++++.++++++++.---------.>++++++++[<---------->-]
<++.>+++++[<++++++++++++++>-]<.++++++++++++++.----------.>++++
+++[<---------->-]<++.>++++++++[<+++++++++++>-]<.>+++[<----->
-]<.>+++[<++++++>-]<..>+++++++++[<---------->-]<--.>+++++++[<
++++++++++>-]<+++.+++++++++++.>++++++++[<----------->-]<++++
.>+++++[<++++++++++++++>-]<.>+++[<+++++++>-]<-.---.++++++.----
---.-----------.>++++++++[<------------>-]<+.---.[-]<<<->[-]>[
-]<<[>+>+<<-]>>[<<+>>-]>>>[-]<<<+++++++++<[>>>+<<[>+>[-]<<-]
>[<+>-]>[<<++++++++++>>>+<-]<<-<-]+++++++++>[<->-]>>+>[<[-]<
<+>>>-]>[-]+<<[>+>-<<-]<<<[>>+>+<<<-]>>>[<<<+>>>-]<>>[<+>-]<
<-[>[-]<[-]]>>+<[>[-]<-]<+++++++++[<++++++<+++++++>>-]>>>[>+>+
<<-]>>[<<+>>-]<[<<<<<.>>>>>-]<<<<<<.>>[-]>[-]++++[<+++++++++>
-]<.>++++[<++++++++++>-]<++.>+++++[<++++++++++>-]<.><+++++..---
------.--------.>>[>>+>+<<<-]>>>[<<<+>>>-]<[<<<<++++++++++++++
.>>>>-]<<<<[-]>++++[<+++++++++>-]<.>++++++++++[<++++++++++>-]<-
-.----------.>+++++++[<---------->-]<.>++++++[<++++++++++++>-]
<.+++..+++++++++++++.>++++++++[<---------->-]<--.>+++++++++[
<++++++++++>-]<--.-.>++++++++[<---------->-]<++.>++++++++[<++
++++++++>-]<++++.------------.---.>+++++++[<---------->-]<+.
>++++++++[<++++++++++++>-]<-.>++[<------------>-]<.+++++++++++
..>+++++++++[<---------->-]<-----.---.+++.---.[-]<<<]
```

Common Lisp

```
(defun bottles (x)
 (loop for bottles from x downto 1
       do (format t "~a bottle~:p of beer on the
wall~@
                     ~:*~a bottle~:p of beer~@
                     Take one down, pass it around~@
                     ~V[No more~:;~:*~a bottle~:p
of~] beer on the wall~2%"
                  bottles (1- bottles))))
```

```
(bottles 99)
```

Scheme

```scheme
(define (sing)
(define (sing-to-x n)
 (if (> n -1)
    (begin
        (display n)
        (display "bottles of beer on the wall")
        (newline)
        (display "Take one down, pass it around")
        (newline)
        (sing-to-x (- n 1)))
    (display "would you wanna me to sing it again?")))
(sing-to-x 99))
```

Haskell

```haskell
main = mapM_ (putStrLn . beer) [99, 98 .. 0]
beer 1 = "1 bottle of beer on the wall\n1 bottle of
beer\nTake one down, pass it around"
beer 0 = "better go to the store and buy some more."
beer v = show v ++ " bottles of beer on the wall\n"
                ++ show v
                ++" bottles of beer\nTake one down, pass
it around\n"
                 ++ head (lines $ beer $ v-1) ++ "\n"
```

Prolog

```prolog
bottles(0).
bottles(X):-
   writef('%t bottles of beer on the wall \n',[X]),
   writef('%t bottles of beer\n',[X]),
   write('Take one down, pass it around\n'),
   succ(XN,X),
   writef('%t bottles of beer on the wall
\n\n',[XN]),
   bottles(XN).

:- bottles(99).
```

Erlang

```
-module(beersong).
-export([sing/0]).
-define(TEMPLATE_0, "~s of beer on the wall, ~s of beer.~nGo to the store and buy some
more, 99
bottles of beer on the wall.~n").
-define(TEMPLATE_N, "~s of beer on the wall, ~s of beer.~nTake one down and pass it
around, ~s of
beer on the wall.~n~n").

create_verse(0)      -> {0, io_lib:format(?TEMPLATE_0, phrase(0))};
create_verse(Bottle) -> {Bottle, io_lib:format(?TEMPLATE_N, phrase(Bottle))}.

phrase(0)      -> ["No more bottles", "no more bottles"];
phrase(1)      -> ["1 bottle", "1 bottle", "no more bottles"];
phrase(2)      -> ["2 bottles", "2 bottles", "1 bottle"];
phrase(Bottle) -> lists:duplicate(2, integer_to_list(Bottle) ++ " bottles") ++
[integer_to_list(Bottle-1) ++ " bottles"].

bottles() -> lists:reverse(lists:seq(0,99)).
```

Erlang (cont.)

```erlang
sing() ->
    lists:foreach(fun spawn_singer/1, bottles()),
    sing_verse(99).

spawn_singer(Bottle) ->
    Pid = self(),
    spawn(fun() -> Pid ! create_verse(Bottle) end).

sing_verse(Bottle) ->
    receive
        {_, Verse} when Bottle == 0 ->
            io:format(Verse);
        {N, Verse} when Bottle == N ->
            io:format(Verse),
            sing_verse(Bottle-1)
    after
        3000 ->
            io:format("Verse not received - re-starting singer ~n"),
            spawn_singer(Bottle),
            sing_verse(Bottle)
     end.
```

3. Review of preliminary concepts

- Mapping of language features to assembly/machine code
- Parameter passing
- Activation records
- Can deal with moderately-sized programs
- Familiar with data structure concepts and applications
- Competence with recursion

Mapping of language features to assembly/machine code

- Machine code is composed of digital binary numbers and looks like a very long sequence of zeros and ones (01001000 01100101 01101100 01101100)
- This is read by the CPU and it understands only machine code, hence any program we write has to be converted to machine code first.
- Machine language is too obscure and complex for use in software development.
- So, low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

```
1: MOV eax, 3
MOV ebx, 4
ADD eax, ebx, ecx
```

Mapping of language features to assembly/machine code (contd)

- Java, C++, Python are all examples of high level programming languages.
- These are converted to low level machine readable code with the help of compilers.
- A compiler is a computer program that translates a program written in a high-level language to the machine level language of a computer.
- Everything is translated and compiled once by the compiler, and then can be used many times afterwards

Mapping of high level language to assembly:
https://cs.brynmawr.edu/Courses/cs240/fall2013/highlevelprogramming.html

Parameter passing

Types of parameters:

- Formal Parameter: A variable and its type as they appear in the prototype of the function or method.
- Actual Parameter: The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Pass By Value:

- There are two copies of parameters stored in different memory locations. One is the original copy and the other is the function copy.
- Changes made to formal parameter do not get transmitted back to the caller.

Pass by reference:

- Both the actual and formal parameters refer to the same locations.
- Changes made to formal parameter do get transmitted back to the caller through parameter passing.

Parameter passing

```
#include<stdio.h>
void getDoubleValue(int *F){
   *F = *F + 2;
   printf("F(Formal Parameter) = %d\n", *F);
}
int main(){
   int A = 8;
   getDoubleValue(&A);
   printf("A(Actual Parameter) = %d\n", A);
   return 0;
}
```

```
Output-
F(Formal Parameter) = 10
A(Actual Parameter) = 10
```

Call by Reference

```
#include<stdio.h>
void getDoubleValue(int F){
   F = F+2;
   printf("F(Formal Parameter) = %d\n", F);
}
int main(){
   int A=8;
   getDoubleValue(A);
   printf("A(Actual Parameter) = %d\n", A);
   return 0;
}
```

```
Output -
F(Formal Parameter) = 10
A(Actual Parameter) = 8
```

Call by Value

Activation records

- An activation record is a contiguous block of storage that manages information required by a single execution of a procedure.
- When you enter a procedure, you allocate an activation record, and when you exit that procedure, you de-allocate it.
- If a procedure is called, an activation record is pushed into the stack, and it is popped when the control returns to the calling function.
- Activation Record includes some fields which are – Return values, parameter list,saved machine status, local data, temporaries etc.

Activation records

- Local variables: hold the data that is local to the execution of the procedure.
- Temporary values: stores the values that arise in the evaluation of an expression.
- Machine status: holds the information about the status of the machine just before the function call.
- Access link (optional): refers to non-local data held in other activation records.
- Control link (optional): points to activation record of caller.
- Return value: used by the called procedure to return a value to calling procedure
- Actual parameters

Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
- There are two types of cases in recursion i.e. recursive case and a base case.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.
- We perform the same operations multiple times with different inputs and in every step, we try smaller inputs to make the problem smaller.
- It uses less number of code lines and hence the code looks shorter and cleaner.

Recursion example

```
#include <stdio.h>

int fibonacci(int i) {
   if(i == 0) {
      return 0;
   }
   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main(void) {
   int i;
   for (i = 0; i < 6; i++) {
      printf("Term %d = %d\n", i+1, fibonacci(i));
   }
   return 0;
}
```

Output -

```
Term 1 = 0
Term 2 = 1
Term 3 = 1
Term 4 = 2
Term 5 = 3
Term 6 = 5
```

4. Introductory lecture

Why study programming languages?

- Increase your capacity to express ideas
- Improve ability to choose appropriate language
- Reduce time to learn new languages
- Understand the significance of language implementation
- Better utilize known languages
- Advance the collective knowledge of computing

4. What languages are used in different domains?

- Scientific Computing:
    - Large numbers of floating point computations
    - FORTRAN, Matlab, R, Python
- Business Applications:
    - Produce reports, use decimal numbers and characters
    - COBOL
- Artificial Intelligence:
    - Symbols rather than numbers manipulated; use of linked lists
    - LISP, (lately: C++/Python)
- Web Software: Java, JS, TS, PHP

4. Programming Language Paradigms

(Switch over to Professor's slides)

- *Imperative*
- *Functional*
- *Logical*

Programming language paradigms

High-level programming languages can be grouped into three categories (more or less):

Imperative: a program is a sequence of commands that modify program state (C, C++, Fortran, Java)

Functional: a program is a composition of math-like functions that don’t modify state when evaluated (Haskell, Ocaml, Lisp if done right)

Logic: a program is a collection of rules that constrain the search for a solution to a problem, also viewed as proving a theorem (Prolog)

Programming language paradigms

And maybe there’s a fourth category:

Object-oriented: a program embodies the concept of “objects” that encapsulate the data and the methods acting on the data (C++, Java, Smalltalk)

Note that object-oriented languages like Java and C++ are imperative languages with useful organizing constructs

Do you have a “favorite” language?

Why is it your favorite?

Language design criteria

What attributes might we want in a programming language?

Efficiency: how efficiently can a program be translated? Compiled?

Expressiveness: is it easy to write complex procedures and structures?

Uniformity: do similar things behave in a similar way?

Simplicity: are the underlying concepts simple?

Readability: is a program easy to read? (writeability too)

Environment: does the language support efficient compilers, interpreters, editors, debuggers, testing and maintenance tools, etc.

Cost: what does it cost to train programmers in this language? What's the cost of maintaining programs in this language?,,,

These are just some of the attributes.

Language design criteria

These criteria are often in conflict with each other.

Balancing these trade-offs leads to new and different programming languages.

For example…

Design goals for Lua

Lua is a scripting language. It is possibly the leading scripting language for video games everywhere. It is also used in embedded devices like set-top boxes and TVs, and in applications like Photoshop Lightroom and Wikipedia.

From “A Look at the Design of Lua” by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, in Communications of the ACM, vol 61, no 11, November 2018.

Design goals for Lua

Simplicity – offer only a few powerful mechanisms that can address several different needs, instead of myriad specific language constructs, each tailored for a specific need. The reference manual is small, approximately 100 pages covering the language, its standard libraries, and the API with C.

Small size – the entire implementation consists of 25,000 lines of C code; the binary for 64-bit Linux is 200k bytes. Small is important for portability and embeddability (see next slide).

Design goals for Lua

Portability – Lua is implemented in ISO C and runs in virtually any system with as little as 300k bytes of memory.

Embeddability – designed from the beginning to interoperate with other languages, both by extending – allowing Lua code to call functions written in another language – and by embedding – allowing ”foreign” code to call functions written in Lua.

4. Influences on Language Design

- **Computer Architecture**
  - Languages are developed around the prevalent computer architecture, known as the von Neumann architecture
- **Programming Methodologies**
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages
    - ex. C is not OO, so C++ was created to address this issue

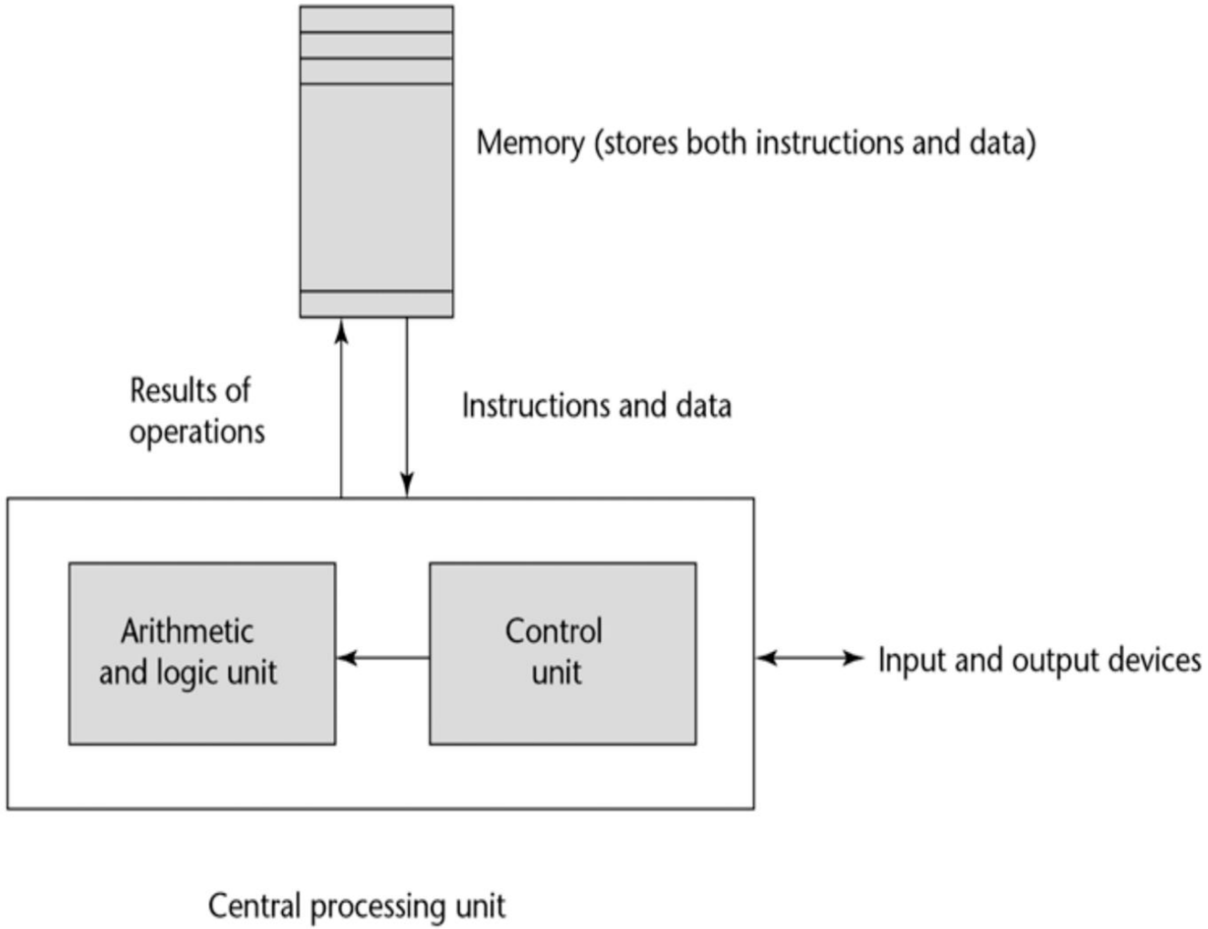
4. Computer Architecture and Language Design

Well-known computer architecture: **Von Neumann**

- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages:
    - Variables model memory cells
    - Assignment statements model data piping
    - Iteration is efficient

Figure 1.1

The von Neumann computer architecture



4. Computer Architecture and Language Design

Fetch-execute-cycle (on a von Neumann architecture computer):

```
initialize the program counter
repeat forever
    fetch the instruction pointed by the counter
    increment the counter
    decode the instruction
    execute the instruction
end repeat
```

4. Computer Architecture and Language Design

Backus 1977 ACM Turing Award Lecture:

“… and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube **the Von Neumann bottleneck**. The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear…”

4. Von Neumann Bottleneck

- Connection speed between a computer’s memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a bottleneck
- Known as the von Neumann bottleneck; it is the primary limiting factor in the speed of computers

4. Understanding Language

(Switch over to Professor's slides)

Understanding Language

Any language, whether it's a natural language like English or an unnatural language like C, is a means of communicating concepts or ideas between people, and in the case of programming languages it's also a means of communication between a person and a computer.

Understanding Language

Ultimately, understanding a sentence or a paragraph in English or a line of code or a whole procedure in C comes down to performing some sort of analysis on the "input stream" of information and determining the meaning.

Understanding Language

Ultimately, understanding a sentence or a paragraph in English or a line of code or a whole procedure in C comes down to performing some sort of analysis on the "input stream" of information and determining the meaning.

What kinds of knowledge must we have to understand a sentence/paragraph/line of code/procedure?

Understanding Language

You're all actually experts at this, although you may not realize it. But the simple fact is that if you're a native English (or other natural language) speaker of college age, you probably have at least a 50,000 word vocabulary in your head, and though you couldn't possibly list them all, you can recognize them when you read or hear them. Plus, you have all the complicated rules of your language’s grammar in your head. And, you know how meanings of individual words combine into a meaningful expression, and so on.

Understanding Language

You couldn't possibly generate all possible grammatically correct English sentences, if only because you'd be dead long before you made a dent in the problem...for that matter, the Sun would burn out long before you made that dent.

All that matters is that you can recognize when some English utterance adheres to those rules of English that are somehow stored in your head, and furthermore that you can extract the intended meaning from that utterance.

Syntax and Semantics

The collection of rules that tell you what orderings of words are acceptable in English, or any other language, is called a grammar for the language. The rules define the form or structure or syntax of a language. Knowing the rules of syntax of a language helps us get at the meaning, but syntax alone won't get us all the way there.

Although you may not have given it any thought, you actually know something about the syntax of programming languages already…

Syntax and Semantics

Which of these is right? Meaning is the same but structure is different.

```
if a > b then WriteLn('Condition met')
    else WriteLn('Condition not met');

WriteLn('Condition not met') unless a > b
    then WriteLn('Condition met');
```

Syntax and Semantics

We also need to know about the semantics of the language--the meaning of the individual components, like the words, as well as the meaning of the expression that consists of a particular ordering of those meaningful components.

Syntax and Semantics

So we use both syntax and semantics in understanding an utterance...hey, let's call it a sentence...in some language. They're both useful, but they aren't joined at the hip...we can look at them separately. For example, in English, we can read the sentence "Colorless green ideas sleep furiously" and see instantly that its structure complies with English grammar, but we'd have to stretch a whole lot to come up with any reasonable meaning for the sentence. That is, the sentence is syntactically valid, but semantically void.

Syntax and Semantics

Fortunately, understanding a natural language like English isn’t your problem in this class. All you have to worry about is programming languages, which tend to be smaller, and neater, and more rigidly defined than natural languages.

Why study syntax and semantics?

Formal specifications of syntax and semantics are how programming language designers specify a programming language. They're also how the implementors know how to construct a compiler for that language. They're also how you, the language user, know whether the line of code you've written is legal in that language.

Programming language reference manuals will sometimes include a formal description of the syntax (see next slide).

Just one part of formal Pascal syntax

Expressions and Variables

expression::= simple_expression #(relational_operator simple_expression),
function_identifier::= identifier,
entire_variable::= variable_identifier,
component_variable::= indexed_variable | field_designator,
identified_variable::= pointer_variable "^",
buffer_variable::= file_variable "^",
simple_expression::= O(sign) term #(adding_operator term),
variable_identifier::= identifier,
indexed_variable::= array_variable "[" O(index_expression #("," index_expression)) "]" ,
field_designator::= record_variable "." field_specifier | field_designator_identifier,
pointer_variable::= variable_access,
term::= factor #(multiplying_operator factor),
array_variable::= variable_access,
index_expression::= expression,
record_variable::= variable_access,
field_specifier::= field_identifier,
field_designator_identifier::= identifier,
factor::= variable_access | unsigned_constant | function_designator | set_constructor | expression |
"not" factor, | bound_identifier,
field_identifier::= identifier,
set_constructor::= O(#(member_designator #("," member_designator))),
bound_identifier::= identifier,
member_designator::= expression #(".." expression),
function_identification::= "function" function_identifier,
function_designator::= function_identifier O(actual_parameter_list),

Processing a program

The structure of a compiler

1. Lexical analysis
2. Syntactic analysis (parsing)
3. Semantic analysis
4. Optimization
5. Code generation

The first three steps are analogous to processing a natural language, sort of.

Compiler: software that translates code in one programming language (e.g., C++) into another programming language (e.g., machine code).

Lexical analysis

The goal here is to combine characters from an input stream of text (the program) into symbols. The analyzer is looking for:

- reserved words/keywords: if, while, for, ...
- identifiers: variable names, function names...
- literals: 100, 1.0, “foo”...
- operators: +, -, *, /, ...
- others: (,), {, }, >, =, ...

Lexical analysis

For example, in the case of the English sentence:

This is a sentence.

You’re looking for the smallest unit above letters (i.e., words).

You note:

- the upper case “T” (start of sentence symbol)
- blanks “ ” (word separators)
- the period “.” (end of sentence symbol)

Lexical analysis

With code from a programming languge, a lexical analyzer divides program text into “tokens”:

```
if x == y then z = 1; else z = 2
```

Tokens are:

if, x, ==, y, then, z, =, 1, ;, else, z, =, 2

Lexical analysis

Different languages have different rules for how the different classes of tokens may be formed.

How does one specify token classes?

One uses regular expressions and finite state automata.

One uses tools such as lex and flex.

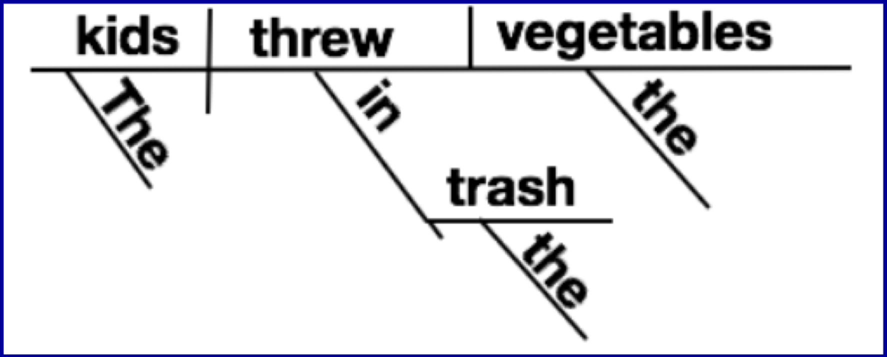
One takes ECS 142 (if it’s ever offered).

Syntactic analysis (parsing)

Once the input stream is successfully converted to a series of tokens, the next step is to determine the sentence structure.

Parse: to resolve a sentence into its component parts and describe their syntactic roles.

Like diagramming sentences?



Syntax

First, some terminology:

- An alphabet is the set of individual characters that can be used in a language.
- A lexeme is the lowest level syntactic unit of a language (e.g., +, ==, do, foo). It's made up of characters from the alphabet. It's a programming language term. In natural language, we tend to call it a word. (Your book equates words and tokens...no big deal.)
- A token is a category of lexemes (e.g., operator, number, identifier). In natural language we call these syntactic categories (e.g, noun, verb, conjunction).

Syntax

More terminology:

- A sentence is a string of lexemes over some alphabet.
- A grammar is a finite set of rules that define the structure or syntax of a language.
- A language is the set of all sentences (possibly an infinitely large set) that adheres to the syntax.

Syntax

A grammar is an example of a meta-language: a language used to describe another language. Grammars define both recognizers (i.e., things that accept any legal sentence) and generators (i.e., things that create all legal sentences).

Generators are easier to talk about in some cases, so we tend to study them more than recognizers, but recognizers are what compilers are built around, so we'll see those too.

A little bit of English

On the next slide is a grammar for a tiny subset of English. It consists of a handful of rules and a handful of symbols. Each rule (or production) has a left-hand side and a right-hand side, separated by an arrow that points to the right.

A little bit of English

```
S    -> NP VP
NP   -> ART NOUN | ART ADJ NOUN
VP   -> VERB NP
ART  -> a | the
NOUN -> dog | frog
VERB -> ate | squashed
ADJ  -> big | green | hairy
```

Everything that's printed in capitals is called a non-terminal symbol. Everything that's printed in lower case is called a terminal symbol. A given rule shows how a single non-terminal symbol can be replaced by (or produces) the symbols on the right-hand side.

A little bit of English

Thus

```
S     -> NP VP
```

says that the S symbol (for sentence) can be replaced by the symbol NP (for noun phrase) followed by the symbol VP (for verb phrase).

Context free grammar

This type of grammar is called a context-free grammar (CFG). Why? Each rule has exactly one non-terminal symbol and no terminal symbols on the left-hand side.

As a result, how the non-terminal symbol on the left-hand side is replaced by symbols on the right-hand side is independent of the symbols that might appear adjacent to the non-terminal on the left-hand side.

That is, the replacement is not affected by the context in which the non-terminal appears.

Context free grammar

In studying programming language (PL) syntax, we'll look mostly at context-free grammars (CFGs), because they're powerful enough to describe the syntax of most PL stuff that we need, but simple enough to make computing with them relatively efficient.

Context free grammar

There are more powerful grammars, called context-sensitive grammars, but getting programs to deal with them is pretty hard, and in programming-language-land we typically don't need the power they provide.

There are also simpler grammars, called regular grammars, but they're not sufficient to describe the syntax of a reasonable programming language.

Context free grammar

Technically speaking, a CFG has four parts:

- A set of terminals (tokens)
- A set of non-terminals (NT)
- A set of production (or grammar) rules
- A start non-terminal (start symbol, like S for sentence in the previous example)

The tiny grammar, revisited

```
S    -> NP VP
NP   -> ART NOUN | ART ADJ NOUN
VP   -> VERB NP
ART  -> a | the
NOUN -> dog | frog
VERB -> ate | squashed
ADJ  -> big | green | hairy
```

Everything that's printed in capitals is called a non-terminal symbol. Everything that's printed in lower case is called a terminal symbol. A given rule shows how a single non-terminal symbol can be replaced by (or produces) the symbols on the right-hand side.

4. Derivation - Simple Grammar

Generate the following sentence using the grammar rules:

the big dog ate a frog

Grammar:

```
S -> NP VP
NP -> ART NOUN | ART ADJ NOUN
VP -> VERB NP
ART -> a | the
NOUN -> dog | frog
VERB -> ate | squashed
ADJ -> big | green | hairy
```

4. Derivation - Simple Grammar

Generate the following sentence using the grammar rules:

the big dog ate a frog

Derivation:

```
S -> NP VP
 -> ART ADJ NOUN VP
 -> the ADJ NOUN VP
 -> the big NOUN VP
 -> the big dog VP
 -> the big dog VERB NP
 -> the big dog ate NP
 -> the big dog ate ART NOUN
-> the big dog ate a NOUN
-> the big dog ate a frog
```

Derivations

We can use the grammar to generate any legal sentence in the language. Here's one example of a derivation: a trace of how a sentence is constructed in the language, beginning with the top-level rule (in this case, it's `S -> NP VP`):

```
S -> NP VP
  -> ART ADJ NOUN VP
  -> the ADJ NOUN VP
  -> the big NOUN VP
  -> the big dog VP
  -> the big dog VERB NP
  -> the big dog ate NP
  -> the big dog ate ART NOUN
  -> the big dog ate a NOUN
  -> the big dog ate a frog
```

```
S    -> NP VP
NP   -> ART NOUN | ART ADJ NOUN
VP   -> VERB NP
ART  -> a | the
NOUN -> dog | frog
VERB -> ate | squashed
ADJ  -> big | green | hairy
```

Derivations

Here’s another example of a derivation for the same sentence:

```
S -> NP VP
  -> NP VERB NP
  -> NP ate NP
  -> NP ate ART NOUN
  -> NP ate a NOUN
  -> NP ate a frog
  -> ART ADJ NOUN ate a frog
  -> the ADJ NOUN ate a frog
  -> the big NOUN ate a frog
  -> the big dog ate a frog
```

```
S    -> NP VP
NP   -> ART NOUN | ART ADJ NOUN
VP   -> VERB NP
ART  -> a | the
NOUN -> dog | frog
VERB -> ate | squashed
ADJ  -> big | green | hairy
```

Derivations

We can display the result of the derivation graphically using something called a parse tree (sort of like the sentence diagram from a previous slide):

$$\begin{tikzcd}
 & & & \text{S} \arrow[dll] \arrow[dr] & & & \\
 & \text{NP} \arrow[dl] \arrow[d] \arrow[dr] & & & \text{VP} \arrow[d] \arrow[dr] & & \\
\text{ART} \arrow[dd] & \text{ADJ} \arrow[dd] & \text{NOUN} \arrow[dd] & & \text{VERB} \arrow[dd] & \text{NP} \arrow[d] \arrow[dr] & \\
 & & & & & \text{ART} \arrow[d] & \text{NOUN} \arrow[d] \\
\text{the} & \text{big} & \text{dog} & & \text{ate} & \text{a} & \text{frog}
\end{tikzcd}$$

Note that the order of terminals in the parse tree is exactly the same as that in the last line of the previous derivations. Order is everything. (Some folks say this is really how you "diagram a sentence". It's not how I learned it when I was a kid.)

Derivations

The grammar we have been working with produces a finite set of sentences. But by adding just a few more rules, we can come up with a grammar that produces an infinite number of legal (though possibly not very meaningful) sentences. Here are the rules. How can you produce an infinite number of sentences by adding a finite number of rules??

```
NP   -> ART NOUN | ART ADJ NOUN | NP PP
PP   -> PREP NP
PREP -> in | on | under | over
```

Derivations

```
S    -> NP VP
NP   -> ART NOUN | ART ADJ NOUN | NP PP
VP   -> VERB NP
PP   -> PREP NP
ART  -> a | the
NOUN -> dog | frog
VERB -> ate | squashed
ADJ  -> big | green | hairy
PREP -> in | on | under | over
```

Derivations

```
S    -> NP VP
NP   -> ART NOUN | ART ADJ NOUN | NP PP
VP   -> VERB NP
PP   -> PREP NP
ART  -> a | the
NOUN -> dog | frog
VERB -> ate | squashed
ADJ  -> big | green | hairy
PREP -> in | on | under | over

NP   -> NP PP
     -> NP PREP NP
     -> NP PREP NP PP
     -> NP PREP NP PREP NP
     -> NP PREP NP PREP NP PP
     -> NP PREP NP PREP NP PREP NP
     -> NP PREP NP PREP NP PREP NP PP…
```

Derivations

Technically speaking, again, the derivation of a string by a grammar...

- Begins with the starting non-terminal symbol
- Proceeds by repeatedly replacing a non-terminal with the right-hand side of one of its rules until only terminals remain

The language L(G) of a grammar G is the set of all strings derivable from the starting non-terminal of G.

Questions?

5. A quick Java tutorial

- Java is an Object-Oriented Programming Language - Focuses on reusable classes and objects.
- Platform-Independent: Write code once, run on different platforms.
- Robust and reliable language for various application domains.
- Rich Standard Library: Extensive built-in tools for various tasks.
- Its applications include web and mobile app development, embedded systems among others.
- The Java codes are first compiled into byte code. Then the byte code runs on Java Virtual Machine (JVM) regardless of the underlying architecture.
- Learning Java opens doors to diverse career opportunities.
- A valuable skill in the dynamic world of software development.

5. A quick Java tutorial

Learn Java in 14 minutes: https://www.youtube.com/watch?v=RRubcjpTkks