Prerequisites

Lecture #00

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

Topics

- "**Experience in an imperative programming language** (C, C++, Java, Python, etc.). The course will be taught in Python, so familiarity is encouraged. However, confidence in another language will suffice, as no specific python features will be important"
- "Knowledge of **basic programming in any assembly language**"
- "Knowledge of **trees, linked lists, basic hash tables, basic recursive tree traversal**"
- "Knowledge of **basic Unix, Bash usage**. Provided course materials assume that the compiler will be built and run on a Linux target, so usage of the `cs1.utdallas.edu` server or a physical Linux machine is required, unless students choose to not use the project base"
- "Knowledge of **git**"

Imperative Programming

I cannot give a succinct tutorial on how to learn imperative programming, this knowledge is assumed. However, knowledge of any imperative programming language that supports the following language features is sufficient to write the code necessary to complete this course.

- Conditionals
- Loops
- Functions
- Simple classes

If following along with the ECCO compiler project base, familiarity of the C programming language is encouraged. C is fairly simple, the only features I would consider "esoteric" in comparison to more modern languages are pointers and macros. Any online tutorial will suffice to learn about these.

Some Python Syntax

```
# Conditional statement, logical operators
if cond1 and cond2 or cond3 and not cond4:
    pass

# Loops (both print out 0-9 with newlines
for i in range(10):
    print(i)

x = 0
while x < 10:
    print(x)
    x += 1
```

Some Python Syntax

```python
# Functions, list/tuple accesses, type hinting
def dot_prod(u: tuple[int, int], v: tuple[int, int]):
    return (u[0] * v[0]) + (u[1] * v[1])

# Prints 11
print(
    dot_prod((1, 2), (3, 4))
)
```

Some Python Syntax

```
# Class definition
Class Dog:
    # Constructor
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age

    # Class methods
    def speak(self):
        # Format string
        print(f"Bark! I'm {self.name}!")

beth = Dog("Beth", "Golden Doberman", 3)
beth.speak() # Prints "Bark! I'm Beth!"
```

Assembly Programming

ECCO will compile C to LLVM-IR, a pseudo-assembly language that is optimized by the `clang` compiler. It is very different from common assembly languages like x86, ARM, or MIPS, but is grounded in similar concepts:

- Registers
- Instructions
- Jumps
- Stack Memory

Assembly Programming - Registers

Registers are tiny (less than 100 bits) spaces on CPUs that store data for the CPU to operate on. This is just enough space to store a character or a pretty large number.

Some registers are general-purpose, so can be used for arbitrary computation of whatever a programmer or compiler wants to accomplish. Some registers are specific purpose, i.e. the MIPS `$zero`: a read-only register that always contains the number 0, program counter registers that contain the current instruction offset of a program, or stack pointer registers that keep track of memory stuff.

Assembly Programming - Instructions

High-level languages have keywords that have special meanings, like `for`, `def`, `and`, etc. Assembly languages use mnemonics or acronyms to denote a small operation on data - usually there are no high-level structures at an assembly languages, concepts like classes and functions are just abstractions. Consider `subu` for "Subtract (Unsigned)" or `bne` for "Branch if Not Equal" in MIPS.

Instructions typically take registers and/or constant values as input and output arguments. For example, `ADD r2,r1,r3` in ARM translates to "add the contents of registers 1 and 3, then stores the sum into register 2. In MIPS, `addi $t2,$t1,64` adds the value 64 to the contents of temporary register 1, then stores the result in temporary register 2.

Assembly Programming - Jumps

In high-level programming, we are very familiar with structures like loops and if-else statements. These are abstractions that can be reduced to conditional "jumps" or "branches". The following code samples in Python and MIPS are equivalent:

```python
if x > 100:
    x = 0
else:
    x = 50
```

```
# t0 = x, t1 = 100
  ble    $t0, $t1, false_clause
true_clause:
  li     $t0, 0                        # x = 0
  j      end_clause                    # skip false clause
false_clause:
  li     $t0, 50                       # x = 50
end_clause:
```

Assembly Programming - Jumps cont.

Jumps are also how loops work. The following two Python loops are structurally equivalent to the following MIPS sample (for loops are a special case of while loops!):

```
x = 0
for i in range(10):
    x = i

x, i = 0, 0
while i < 10:
    x = i
    i += 1
```

```
# t0 = x, t1 = i, t2 = 10
  li    $t0, 0
  li    $t1, 0
loop_head:
  move  $t0, $t1
  addi  $t1, 1
  blt   $t1, $t2, loop_head
```

Assembly Programming - Stack Memory

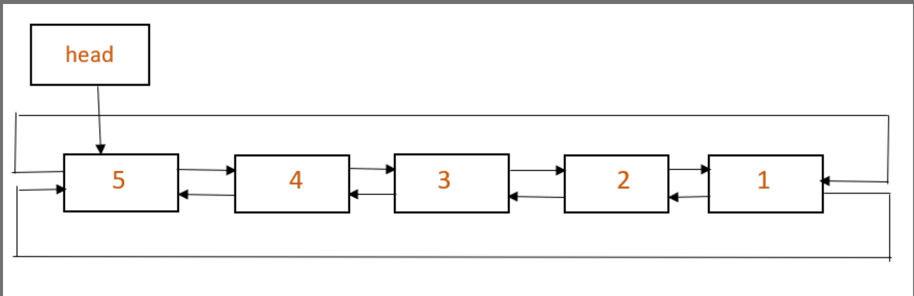
CPUs usually have a register called a "stack pointer", which points to the expanding end of stack memory in RAM. Whenever we have data that is too big to fit in a register, it ends up on the stack.

In a high level language, when functions are called inside of each other, the current register contents are typically pushed onto stack memory so that the newly-called function can use the registers without worrying about overwriting important data. When the child function terminates, the parent function will "pop" the data back from the stack into the appropriate registers.

In this way, we can have a very large amount of functions call each other, at least until stack memory runs out.

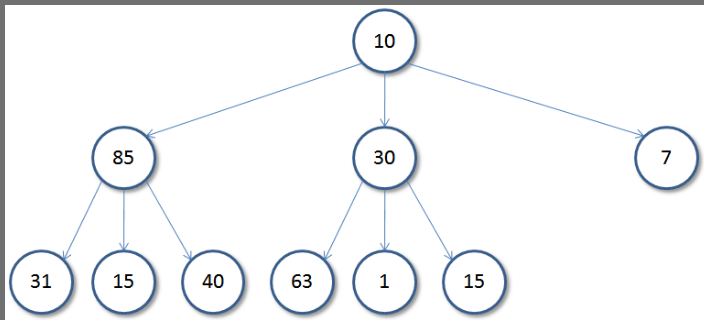
Data Structures - Linked Lists

A Linked List is a (usually) bidirectional network in which each node has exactly 0 or 1 child nodes (successors) and exactly 0 or 1 parent nodes (ancestors).



Data Structures - Trees

A Tree is an oriented, (usually) bidirectional, hierarchical network of parent and child nodes. This means that there exist a set of nodes, some who are parents of multiple other nodes. A node can only be the child of one parent. This image is of a "ternary" tree (each node can have up to 3 children).



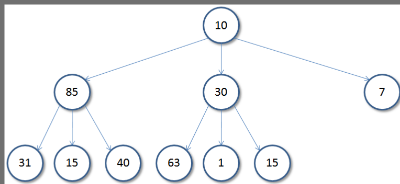
Data Structures - Tree Traversal

"Traversal" of a tree describes a set of algorithms that move around the tree network and perform operations using the data stored at each node and the relations between nodes. For our cases, traversal algorithms will visit every node in the tree.

The two primary tree traversals are **Breadth-First Traversal** (BFS) and **Depth-First Traversal** (DFS).

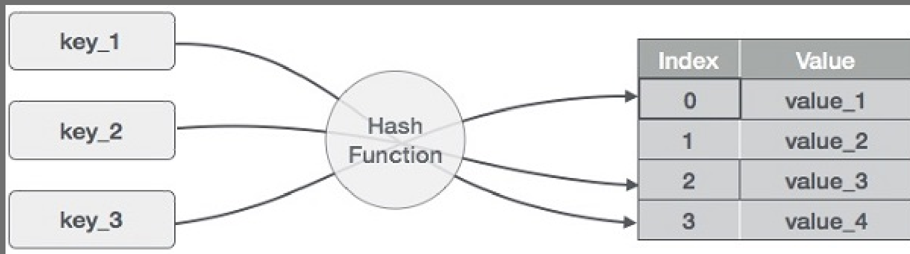
DFS of the below image: 10 85 31 15 40 30 63 1 15 7

BFS of the below image: 10 85 30 7 31 15 40 63 1 15



Data Structures - Hash Tables

A Hash Table is a key-value-type structure in which keys are passed through a "hash function", a one-way encryption scheme that takes in various kinds of data and outputs an integer that represents the row index of corresponding data (values) in the table. Values can be any kind of data, including a Linked List (we will utilize this tactic later).



Unix and Bash usage

You should know the very basics of Unix usage, such as how to run Bash scripts. Our ECCO project base has a "scripts" file, which takes arguments to perform different actions. For example, `./scripts run test_file` will start up our compiler and pass in a file called "test_file" to it, then execute the generated assembly code.

`./scripts all test_file` will format and lint your compiler source code, and then compile and execute `test_file`.

If you do not have access to a Unix-based machine (you do if you're a UTD student!), I expect that you understand Docker well enough to use the provided Dockerfile in a Windows environment to run your code. Or, I expect that you can set up WSL and use it accordingly.

Git usage

If this will be your first time using Git, I am excited to welcome you into the next phase of your career as a software developer. Git is arguably the most impactful software development tool since the first text editor was designed.

Git is a versioning system, allowing you to track changes in your source code over time. This is helpful if you need to make a mistake and revert to a version of your code that works, or to see where bugs were introduced in the past.

Git usage cont.

Your "working directory" consists of the compiler source code and whatever other files are adjacent to it. Any changes performed in the working directory are temporary and can be reverted after a warning.

The "staging area" is where you temporarily put your changes when you're almost ready to cement them. This is accomplished by running the command '`git add [list of filenames you want to add to staging area, separated by spaces]`'.

When you're ready to cement your changes, you can run '`git commit -m "[descriptive message of your major changes]"`'. These will add your changes to the "committed files" area of the repository. Commits *can* be altered after running `git commit` but it's non-trivial, so you can treat this as a permanent action for many intents and purposes.

Remote Git usage

Sometimes you want to back up your git repository to an external server, maybe to be safe, or to collaborate with others. In those cases, you can use a service like GitHub or Gitlab. These websites let you remotely host git repositories.

If there are changes on your local repository that you want to add to the remote repository, you can run '`git push origin main`'. Here, "origin" is the standard and default pseudonym for any remote server that hosts git instances. Also, "main" is the name of a "branch".

Git lets you split your development across multiple versions of your code (branches) at the same time. These branches can be merged together such that multiple people can work on disjoint parts of the software in tandem and then combine their work with a deterministic way to resolve any conflicting changes they might have made. You likely won't need to work on multiple branches for this course.