Welcome to Computer Team!

and Computer Systems I: Introduction

Noah Singer

Montgomery Blair High School Computer Team

September 14, 2017

Overview

Section 1: Introduction

Computer Team

Computer Team is...

- A place to hang out and eat lunch every Thursday
- A cool discussion group for computer science
- An award-winning competition programming group
- A place for people to learn from each other
- Open to people of all experience levels

Computer Team isn't...

- A programming club
- A highly competitive environment
- A rigorous course that requires commitment

Structure

- Four units, each with five to ten lectures
- Each lecture comes with notes that you can take home!
- Guided activities to help you understand more difficult/technical concepts
- Special presentations and guest lectures on interesting topics

Conventions

Vocabulary terms are marked in **bold**.

Definition

Definitions are in boxes (along with theorems, etc.).

Important statements are highlighted in italics.

Formulas are written in

$$\sum f(a^n c_y) \int m^{a^t} h$$

Section 2: Units

Unit I: Computer Systems

Fundamental question: *How do computers function at a low level?*

Lectures:

1. Introduction to Computer Team!
2. Kernels and Operating Systems
3. Networks
4. CPU Architecture
5. Threading and Parallelization
6. Compilers

Unit II: Theory of Computation

Fundamental question: *What is a computer, and what can it do?*

Lectures:

1. Automata
2. Context Free Grammars and Parsing
3. Guided Activity: Lambda Calculus
4. Turing Machines and Computability
5. Guided Activity: Reduction and P vs. NP
6. Computability and the Complexity Hierarchy
7. Randomization and Probabalistic Algorithms

Unit III: Machine Learning+

Fundamental questions: *How can we get computers to solve complex problems?*

Lectures:

1. Introduction to Machine Learning
2. Guided Activity: Neural Networks
3. Advanced Neural Networks
4. Natural Language Processing
5. Fourier Analysis and Signal Processing

Unit X: Algorithms and Contest Preparation

Fundamental question: *What is the best way to solve a given problem?*

Lectures:

1. Graph algorithms
2. Dynamic programming
3. Computational geometry
4. Greedy algorithms
5. Divide-and-conquer and recursive algorithms
6. Online algorithms

Section 3: Computer Systems

Basic constructs

Definition (Data)

Information represented by a sequence of symbols.

- The most basic unit of data in computer science is the **bit**, which is either a 1 or a 0.
- A single bit represents whether a single electrical signal is on (1) or off (0).

Definition (Program)

A series of **instructions** that tell a computer how to manipulate data to transform some defined **input** to a defined **output**.

Fixed- vs. stored-program computers

- Early computers were **fixed-program computers** that could only execute a single, predefined program, hardwired into the computer's circuitry.
- The theory of the **Universal Turing machine** demonstrated that a single computer could execute all programs.
- Important pioneers, like John Von Neumann, developed the **stored-program computer**, which *treats programs themselves as data*.

Code execution

- The random access memory, or **RAM**, stores data, organized into sequential blocks of eight bits called **bytes** assigned location numbers called **addresses**.
- The central processing unit, or **CPU**, executes code sequentially, which is stored as data in RAM.
- Each instruction is encoded as an **opcode** representing the operation being performed and zero or more operands.
- A single instruction can do one or more things, including:
  - Arithmetic and logic
  - Comparisons
  - Control flow
  - Memory access
  - Peripheral access

Section 4: Computer Parts

CPU

- Performs computations quickly
- Keeps track of which code to execute and executes it sequentially
- Accesses memory to store and retrieve data
- Interacts with other hardware devices (**peripherals**)
- Protects sensitive data and enforces security procedures
- Contains **registers** for extremely fast data access
- Executes a fixed sequence of instructions on machine startup from the ROM

RAM

- Stores data organized into bytes
- Organized both by **physical addresses** and **virtual addresses** which are mapped to physical addresses
- **Volatile**: does not persist after power loss

Other components

Permanent storage

- Non-volatile unlike RAM but much slower
- May be removable or non-removable
- Ex.: Hard drives, floppy discs, USB drives, etc.

I/O peripherals

- Enable interaction with users
- Often connected to the CPU through *buses* like USB or PCI
- Ex.: keyboard, mouse, speakers, microphone, printer, etc.

Other components

Graphics processing unit (GPU)

- Render the computer's visual output for interaction with the user
- Generally features hardware acceleration for e.g. 3D graphics
- Often faster than the CPU at raw computations!

Power supply

- Provides a steady power source for the components
- Temporarily insulates against power fluctuations

Motherboard

- Allows all the components to communicate with each other

Conclusion

We'll learn about all this in much more detail in the coming weeks, but for now...

Welcome to Computer Team!

Computer Systems II: Kernels and Memory Management

Noah Singer, George Klees

Montgomery Blair High School Computer Team

September 29, 2017

Overview

Section 1: Kernels

Stored-program computers

- Programs compiled into a sequence of instructions called **machine code**
- Stored as simple data in RAM
  - Very long sequence of **bytes** (8 **bits**)
  - Each byte has a numerical **memory address**
  - CPU can **load and store** data from RAM
    - CPU also has much faster, smaller memory called **registers**
- Data executed as code by CPU

Operating systems

- Intermediary between hardware and applications
- Creates a high-level interface for application developers
- Controls access to hardware and enforces security procedures
- Often separated into core **kernel** and external **drivers**
    - Drivers are often used to interface with specific hardware or accomplish a specific task, and vary from computer to computer depending on hardware and setup
    - Kernel accomplishes core tasks regardless of specific hardware

Operating modes

- When the CPU first loads, any executed code can access all memory, CPU functions, and hardware devices
  - On a secure and reasonable system, applications must be **sandboxed**
- Modern CPUs have two modes that code can execute in
  - **Supervisor**: unrestricted access to resources, only granted to kernel which accomplishes core OS functions
  - **User**: access restricted to only certain memory, certain devices, etc. (for applications)
- Special instructions called **syscalls** allow a user mode application to "jump" into the kernel in supervisor mode to accomplish an OS task
  - Read a file from the hard drive
  - Create (fork) a new process

Types of kernels

- **Microkernel**: Only the bare minimum that requires supervisor mode is in the kernel
  - Many OS functions are user mode drivers
  - More secure and elegant
  - Slower, since many switches between user and supervisor modes may be required
- **Monolithic**: Most of the OS is in the kernel
  - Less secure since any vulnerability in the much larger kernel leads to supervisor control over the system
  - Faster and less complicated
- **Hybrid kernel**: Middle ground
  - Some user mode drivers, some supervisor mode drivers

Core kernel/OS functions

- **Memory management**: allocating and controlling memory for applications
- **Task management**: allowing multiple applications to run simultaneously
- Virtual filesystem (**VFS**): access to stored data
  - Many layers of drivers involved: storage drivers, bus drivers, filesystem drivers, etc.
  - Must present a uniform interface for applications
- Device and power management

Sample (simplified) x86 boot process

1. BIOS boot code is loaded from read-only memory (**ROM**), triggering code in hard drive Master Boot Record (**MBR**)
2. MBR triggers code in hard drive partition Volume Boot Record (**VBR**), loading the kernel and boot drivers from the filesystem with a filesystem driver
3. Kernel activates memory manager and enables paging
4. Kernel sets up syscalls and fault handlers
5. Kernel loads VFS using boot drivers and uses VFS to load other necessary drivers
6. Kernel initializes timers, various buses, and other hardware devices
7. Kernel begins multitasking (multicore?), switches into user mode, and loads user login application

Section 2: Memory Management

Memory management

- Every application's memory must be sandboxed from the others
- Each application has a unique **address space** of **virtual memory** mapped by the memory manager to **physical memory**
- When an application executes, it "sees" the address space of virtual memory
- In modern CPUs, memory is organized into medium-scale (often 4096 B) **pages**
- Task of kernel memory manager: for each application, create a correspondence between virtual pages and physical pages

Kernel memory manager

- **Physical memory allocator** allots pages of physical RAM to applications and the OS
- **Virtual memory manager** organizes the address space of applications
- Kernel communicates with paging hardware (memory management unit, or **MMU**) inside of CPU
  - **Page tables** stored in memory; MMU contains a pointer in a special register
  - MMU translates memory accesses from virtual to physical addresses
  - Lookups cached in the translation lookaside buffer (**TLB**)

Page tables

- Stored as an array of **page table entries (PTEs)**, each representing a page
- PTE stores detailed information, allowing for fine control over memory access:
  - Address of page in physical memory
  - Permissions
  - Caching information

Application address space

- Four main regions:
  1. Program code and data
  2. **Heap**: dynamically allocated data growing upwards
  3. Unallocated memory
  4. **Stack**: return addresses/stack frames, local variables, parameters, execution context
- When invalid memory is accessed, MMU throws a **page fault**, halting execution; can be used creatively to great advantage
  - Memory-mapped files
  - Swapping
- OS memory manager must: allocate and free memory, allow shared memory, map and unmap files, etc.

Conclusion

- OS interfaces between hardware and applications
- Code executes in user mode or supervisor mode
- Virtual memory allows kernel to control application address spaces

Computer Systems III: Multitasking and Concurrency

Noah Singer, George Klees

Montgomery Blair High School Computer Team

October 5, 2017

Overview

Section 1: Multitasking

Processes and threads

- **Process**: A high-level task; an instance of a running program, with an address space and access to resources
- **Threads**: A low-level task; code running on the CPU, consisting of the actual execution state (CPU registers, stack, etc.)
- One or more threads per process
- **Multitasking**: CPU switches back and forth rapidly between threads

Program execution

1. Create blank address space
2. Load program from disk into address space
3. Create initial thread with stack
4. Begin executing

Scheduling

- **Scheduler** must select which threads to run and for how long
  - **Cooperative scheduling**: threads can run as long as they want until they yield
  - **Preemptive scheduling**: threads get a fixed amount of time (**quantum**)
- Threads are generally non-malicious; when any thread has no work to do, it often just yields to allow other threads a turn
- Threads often voluntarily yield when waiting for an operation to complete (**blocking**)
  - Most threads spend most of their time waiting for locks and I/O, so cooperative scheduling is common

Prioritization

- Some threads are more important than others
- **High-priority** threads must complete immediately
  - Examples: GUI, some drivers
  - Often simply receive, route, and transmit requests, sleeping almost immediately after awaking
- **Low-priority** threads can be scheduled whenever there is no more important work
  - Examples: Background processes, virus scanners, updates

Task switching

- When a thread exceeds its quantum, it is **preempted** and loses control
  - Triggered by a hardware timer
- Scheduler holds a queue of threads waiting to run
- When a yield or preemption occurs, the scheduler must switch **CPU contexts** (stack and registers)
  1. Save old CPU context
  2. Switch address spaces, etc. to new process
  3. Restore new CPU context

Section 2: Concurrency

Multiple CPUs

- Scheduler also selects which CPU to schedule a thread on
  - CPU load, power management status, temperature, etc.
  - Due to caches and **affinity**, threads are better scheduled on CPUs on which they have already/recently executed
- **Concurrency** issues occur when threads on two CPUs access the same resource simultaneously
  - Or one thread is working, gets preempted, and then another thread accesses the same resource
  - The kernel is one such resource

Locks

- **Spinlocks** simply record whether or not they are locked
  - Order in which threads attempt to acquire lock is irrelevant
  - Threads "spin" (in a while loop) while waiting for lock
- **Granularity**: the amount of code that is protected by locks
  - Too fine: doesn't fix the original issues
  - Too coarse: doesn't scale and difficult to develop/maintain

Atomic operations

- One thread can be interrupted *while* acquiring the spinlock
- Uninterrutible operations; another CPU cannot interfere
- Thread cannot be preempted in the process of accessing a resource
- In multi-step atomic operations, memory bus is locked, slowing down the rest of the system
- **Compare and exchange** compares the value of `var` and `old`, setting `var` to `new` if it is equal to `old` and returning the original value of `var`

Issues with spinlocks

- Threads cannot yield while waiting for spinlocks
- **Deadlock**: two threads both spinning to acquire resources the other thread has
- **Starvation**: a thread is continually outcompeted for locks, so it cannot access necessary resources
- Some more sophisticated locks solve these issues
  - **Ticket locks** á la deli counter or DMV

Computer Systems IV: Computer Engineering

Noah Singer, George Klees

Montgomery Blair High School Computer Team

October 12, 2017

Overview

Section 1: CPU Architecture

Terminology

- Code is stored as binary data (**machine code**)
- We consider simpler **reduced instruction set computing (RISC)** models
  - Sequence of **instructions**
  - Each instruction has a numerical **opcode** and possibly some **operands** (addresses, direct numbers, registers, etc.)
- CPUs are organized in an **architecture**
  - **Instruction set architecture (ISA)**: types and meanings of instructions (e.g. x86)
  - **Microarchitecture**: specific layout of CPU itself and implementations of instructions (e.g. Pentium III)
  - We consider the simple and widespread **reduced instruction set computing (RISC)** ISA

Common instructions

- Memory loads and stores
- Register switches (moves)
- Arithmetic and logic computations
- Comparisons
- Control flow instructions (**branches** and **jumps**)
- Stack instructions
- Empty operations (**nops**)

Speical instructions

- **Vector instructions** allow computations on many values in a large array simultaneously
- **Floating point instructions** extend CPU arithmetic to non-integers using a **floating-point unit (FPU)**
- Syscalls
- **Interrupt** management

Registers

- CPU contains a small amount of specialized memory called **registers**
- **General-purpose registers** store results of computations
  - **Load and store** instructions access memory, transferring to and from GPRs
- Various **special registers** have specific purposes for the CPU
  - **Instruction pointer** contains memory address of currently executing code
  - **Stack pointer** contains memory address of current stack
  - **Link register** contains return address

Code execution

1. CPU retrieves instruction from memory pointed to in instruction pointer (**instruction fetch**)
2. CPU decodes instruction
3. *Optional*: CPU performs the computation
4. *Optional*: CPU reads from memory and/or writes to memory
5. *Optional*: CPU writes results back into registers
6. CPU advances instruction pointer

Essential components of a modern CPU

- **Arithmetic/logic unit (ALU)** performs mathematical computations
- Memory interface containing **memory management unit (MMU)**
- **Instruction fetcher** retrieves instructions from memory, **instruction decoder** translates the opcodes into operations within the CPU
- **CPU clock** is a crystal oscillating at some **clock rate** that controls instruction execution
- **Caches** optimize data access
- **Register file** is small memory containing registers

Caching

- Memory access can take many clock cycles
- CPU often keeps one or more **caches** of memory or other resources
- When a memory read is made, either:
  1. The address is already in the cache, and it is returned immediately (**cache hit**)
  2. The address is not in the cache, so a memory access must be made, and the value is copied into the cache (**cache miss**)

Section 2: Optimization

Instruction-level parallelism

- Often, the results of instructions do not **depend** on each other, so multiple instructions can be executed somewhat simultaneously
- **Scalar processors** execute instructions sequentially, so **superscalar processors** can execute (parts of) multiple instructions at the same time by having e.g. multiple ALUs
- **Very long instruction word (VLIW)** architectures promote instruction-level parallelism in software (generated by the compiler), so that code is structured with very little dependency

Pipelining

- CPU execution is driven by master **instruction clock**
- Instructions can often be divided up into discrete **stages**
  - Example (RISC): instruction fetch, instruction decode, execute, memory access, register writeback
- Instead of executing each instruction's stages individually, instructions progress through the pipeline in stages, increasing **throughput**

Hazards

- Pipeline stages can interfere with each other, causing faults called **hazards**
- **Bubbles** can be introduced to halt all stages until one stage propagates through
- **Data hazards**: the results of one instruction depend on the results of a previous instruction still in the pipeline
- **Structural hazards**: a single component in the CPU is used twice during the same clock cycle
- **Control hazards**: one instruction changes control flow while other instructions have already began to propagate through the pipeline

Speculative execution

- Conditional branches cause the pipeline to halt until the value of the condition is known (expensive)
- CPUs employ **branch prediction** to guess the result of a conditional branch and begin executing it
- When the same condition is evaluated several times, its results can be stored and analyzed to predict a new condition
- If the wrong condition was predicted, incomplete instructions in the pipeline must be “unrolled” and “refilled” (expensive)

Out-of-order execution

- Some CPU resources are costly to use (e.g. memory, certain arithmetic)
- When instructions are independent, they can be reordered to optimize CPU resource usage
- Data dependency can be further reduced by **renaming registers**

Computer Systems V: Networking

Noah Singer

Montgomery Blair High School Computer Team

November 30, 2017

Overview

Section 1: Basics

Motivations

- Single computers are limited (memory, computational power, etc.)
- Networks allow computers to specialize and to increase total capacities
- Additional advantages of distributed computing
  - Redundancy

History

- The **Internet** connects computers across the world, while the **World Wide Web** is a specific system for organizing documents and information on the Internet
- **ARPANET** was the first network to the **TCP/IP protocol**, which underlies the modern Internet, in 1974
  - TCP/IP was invented by Robert Kahn and Vint Cerf
  - Funded by DARPA and the NSF
- Tim Berners-Lee invented the World Wide Web in 1989
  - Organized by **uniform resource locators (URLs)**

Basics

- In the most basic arrangement, a **client** communicates with a **server**
- Client **requests** a **remote service**, server **provides** the service
- Client sends requests, server returns **responses**
- Networks pass units of data called **packets** from source to destination at certain **addresses** on certain **ports**
  - **Header** provides routing information and other important metadata
  - **Payload** is actual data being transmitted
- **Protocols** define how information is transmitted and how interactions take place

Network topologies

- The computers on a network are laid out locally according to some topology, such as:
  - **Star**: one central hub
  - **Tree**: hierarchical structure
  - **Ring**: connected circularly
  - **Mesh**: as many connected together as possible
  - **Bus**: all connected along a single link

OSI model

- The **Open Systems Interconnection (OSI) model** is a standard conceptual design for a computer network
- Seven **layers** building from the lowest to highest level
- Each layer is associated with a **protocol data unit (PDU)** which describes the "quantum" of data that the layer transmits, stripping headers from lower layers

#	Type	Layer	PDU
7	Host	Application	Data
6	Host	Presentation	Data
5	Host	Session	Data
4	Host	Transport	Segment/Datagram
3	Media	Network	Packet
2	Media	Link	Frame
1	Media	Physical	Bit

Physical layer

- Actual binary signals are transmitted
  - Electrical signals in a wire
  - Fiber optic cables
  - Wireless (radio, WiFi, etc.) signals
- Communication channel can either be:
  - **Simplex**: one way only
  - **Half-duplex**: one way at a time
  - **Full duplex**: both ways at a time
- Also includes network topology

Network layer

- The **Internet protocol (IP)** controls routing of data
  - Data is fragmented into **datagrams** to be transmitted in the physical network
  - Each datagram is routed from the source IP to the destination IP
  - The network is divided into many **subnets**

Transport layer

- Arbitrary-size data sequences are transmitted
- Centered around the **Internel protocol suite**
- The **transport control protocol (TCP)** is coupled with IP in **TCP/IP** to send data over the Internet
  - Data is split into **segments**
  - Segments are received in order
  - Network is reliable and error-checking is implemented
  - Packets can be resent when delivery fails
- The **user datagram protocol (UDP)** does not guarantee delivery and is connectionless

Section 2: Application Layer Protocols

DNS

- The **Domain Name System (DNS)** maps domain names to IP addresses
- The entire space of domain names is partitioned into a hierarchical structure of **DNS zones**, which **delegate** name resolution to subzones
- Every name server holds several **resource records** which, for example, define subdomains and map domains to IPs

HTTP

- The **Hypertext Transfer Protocol (HTTP)** transmits specially annotated text, called **hypertext**, in the World Wide Web
- A **user agent** (like a **web browser**) requests a specific resource using HTTP, and the server responds with the requested data, or possibly a **status code**
- The protocol is **stateless**: it maintains no information between requests

Mail

- The **Simple Mail Transfer Protocol (SMTP)** is used to send email messages between mail servers
- User mail applications often use the more advanced **Internet Message Access Protocol (IMAP)** or **Post Office Protocol 3 (POP3)** to access messages

Security

- Many different Internet protocols and services are secured by the **Transport Layer Security (TLS)** protocol
- A client and server use a **handshake** to established a shared secret key to use for private and secure communication
- Key features:
  - **Privacy**: Messages cannot be read in transit
  - **Authentication**: Senders and receivers of messages can be verified
  - **Integrity**: Messages cannot be modified in transit

Webpage sequence

1. The operating system resolves the DNS record of the requested URL to retrieve the server IP
2. The browser sends an HTTP request packet to the server (over TCP/IP) through **router**, **modem**, and **Internet service provider (ISP)**
3. The server processes the request, loads resources, runs server-side code, etc.
4. The server replies to the browser with a status code HTTP 200/OK
5. The browser interprets and renders the returned HTML code

Computer Systems VI: Compilers

Noah Singer, George Klees

Montgomery Blair High School

April 8, 2018

What are Compilers?

Definition (Compiler)

A program that translates code from a **source language** into a **target language**.

- Usually we're dealing with from some **high-level language** (e.g. C, C++) to **assembly language**
- Assembly is then assembled by an assembler

Hello, Compiler World

Here's a simple C program.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello, compiler world!\n");
    return 0;
}
```

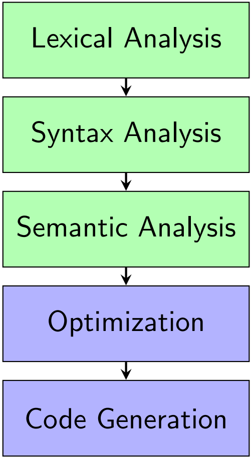
Next, we're going to see what kind of assembly this compiles to.

Hello, Assembly World


```
main:
  sh %rbp
  mov %rsp,%rbp
  sub $0x10,%rsp
  mov %edi,-0x4(%rbp)
  mov %rsi,-0x10(%rbp)
  mov $0x40060c,%edi
  callq 4003f0 <puts@plt>
  mov $0x0,%eax
  leaveq
  retq
```

This begs the question: **What does the compiler do in order to transform code from C to assembly?**

Stages of a Compiler



The five traditional stages of a compiler.

Stages of a Compiler

1. **Lexical analysis.** The source code is split into **tokens** like integers (INT), identifiers (IDEN), or keywords (e.g. IF).
2. **Syntax analysis.** The source code is analyzed for structure and parsed into an **abstract syntax tree** (AST).
3. **Semantic analysis.** Various intermediate-level checks and analyses like **type checking** and making sure variables are declared before use.
4. **Optimization.** At this point, the code is usually converted into some platform independent **intermediate representation** (IR). The code is optimized first platform-independently and then platform-dependently.
5. **Code generation.** Target language code is generated from the IR.

Lexical Analysis

- Programs which perform lexical analysis are called **lexers** for short
- Two main stages:
  1. The **scanner** splits the input into pieces called **lexemes**
  2. The **evaluator** creates tokens from lexemes, in some cases assigning tokens a **value** based on their lexeme
- Whitespace and comments are ignored
- Lexical analysis is considered "solved" since efficient algorithms have been discovered
  - Programs called a **lexer generators** exist which will automatically create lexers
  - Most common lexer generator is called `flex` (new version of `lex`)

Guessing Game

```
printf("Guess a number between 1 and 100!");
int num = 21;

// Read an initial guess and then keep guessing
int guess;
scanf("%d\n", &guess);

while (guess != num)
{
    printf("Guess again!");
    scanf("%d\n", &guess);
}

// They got it right
printf("Good job! You got it!");
```

Guessing Game: Tokens

IDEN<printf> LPAREN STRLIT<Guess a number between 1 and 100!> RPAREN SEMI IDEN<int> IDEN<num> ASSIGN INT<21> SEMI IDEN<int> IDEN<guess> SEMI IDEN<scanf> LPAREN STRLIT<%d\n> COMMA UNARY<&> IDEN<guess> RPAREN SEMI WHILE LPAREN IDEN<guess> BINARY<!=> RPAREN LCURLY IDEN<printf> LPAREN STRLIT<Guess again!> RPAREN SEMI IDEN<scanf> LPAREN STRLIT<%d\n> COMMA UNARY<&> IDEN<guess> ENDWHILE SEMI RPAREN IDEN<printf> LPAREN STRLIT<Good job! You got it!> RPAREN SEMI

Primer on Regular Expressions

Let's take a look at **regular expressions** or **regex**, which are a useful tool for creating lexers. Regular expressions allow programmers and mathematicians to express "patterns" that encompass certain groups of strings.

- + is a unary postfix operator denoting "one or more"
- ? is a unary postfix operator denoting "zero or one"
- * is a unary postfix operator denoting "zero or more"
- (and) can be used to group things together for precedence, just like in normal arithmetic
- [and] can be used for "character classes" (e.g. [0−9])
- | is an infix binary operator denoting "or"

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`
5. Regex to match valid C/Java variable names

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`
5. Regex to match valid C/Java variable names
6. Regex to match only binary strings divisible by three

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`
5. Regex to match valid C/Java variable names
6. Regex to match only binary strings divisible by three
7. `()`, `(())`, `((()))`, `(((())))`, etc.

Regular Expressions for Lexical Analysis

- The lexer is based off of a **lexical grammar** that contains a pattern for each type of token
- Efficient parsing can then be completed using **deterministic finite automata** or **nondeterministic finite automata**
- The evaluator assigns a value to some tokens (e.g. INT tokens) based on their corresponding lexeme

Sample lexical grammar:

IDEN: `[a-zA-Z_][a-zA-Z0-9_]*`
INT: `(\+|-)?[0-9]+`
WHILE: `while`

and more!

Syntax Analysis

- As we've seen, regular expressions and lexical analysis by themselves aren't capable of encompassing the full complexity of programming languages
- For this, we need syntax analysis, also known as **parsing**
- Programs that create parsers are known as **parser generators**, the most popular of which is `bison` (new form of `yacc`); `bison` and `flex` play together very nicely
- Generally parses the stream of tokens into an abstract syntax tree or **parse tree**, difference being that abstract syntax tree doesn't include every detail of source code syntax
- Usually accomplished with a **context-free grammar** (CFG)
- Parsing can be either **bottom-up** or **top-down**
- To understand this, let's take a look at some formal language theory!

Formal Language Theory

- A **language** is a set (finite or infinite) of **words** over some **alphabet** consisting of **letters** Σ
- Each language has **syntax**, which describes how it looks, and **semantics**, which describes what it means
- A language is often defined by some set of rules and constraints called a **grammar**
  - Consists mostly of **productions**, which are rules mapping some **symbols** to the union of one or more strings of symbols
- Example language (this only only has one production): $E \rightarrow (E) \mid E * E \mid E/E \mid E + E \mid E - E \mid \text{INT}$
  - What is this?
- In context-free languages, a symbol can always be replaced using a production, regardless of its context (left hand side is only one symbol)

Guessing Game: Revisited

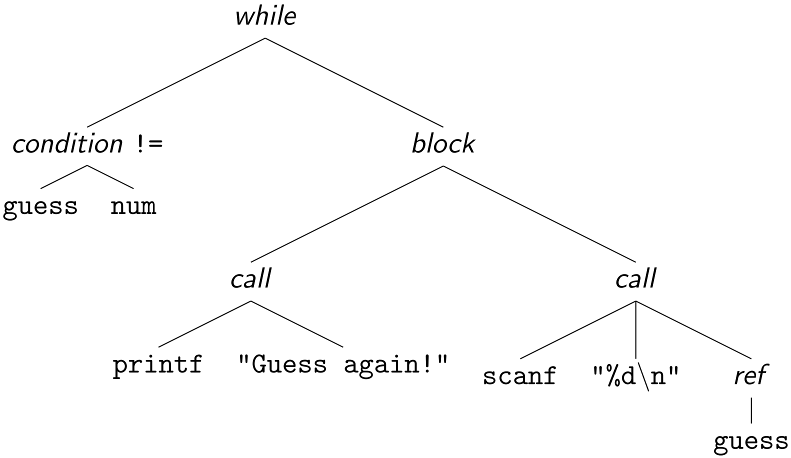

```
printf("Guess a number between 1 and 100!");
int num = 21;

// Read an initial guess and then keep guessing
int guess;
scanf("%d\n", &guess);

while (guess != num)
{
    printf("Guess again!");
    scanf("%d\n", &guess);
}

// They got it right
printf("Good job! You got it!");
```

Guessing Game: Abstract Synax Tree



Guessing Game: Model Grammar

A vastly oversimplified model grammar for C that somewhat works with our example.

$$\textit{while} \to \textit{condition block}$$
$$\textit{condition} \to \textit{expr}\ \mathrm{CMP}\ \textit{expr}$$
$$\textit{expr} \to (\textit{expr}) \mid \mathrm{UNARY}\ \textit{expr} \mid \textit{callexpr}\ \mathrm{BINARY}\ \textit{expr} \mid \mathrm{IDEN} \mid \mathrm{STRLIT}$$
$$\textit{block} \to \textit{statement block}$$
$$\textit{statement} \to \textit{call}\ \mathrm{SEMI} \mid \textit{assign}\ \mathrm{SEMI}$$
$$\textit{call} \to \mathrm{IDEN}\ \mathrm{LPAREN}\ \textit{args}\ \mathrm{RPAREN}$$
$$\textit{assign} \to \mathrm{IDEN}\ \mathrm{IDEN}\ \mathrm{ASSIGN}\ \textit{expr}$$
$$\textit{args} \to \textit{expr} \mid \textit{expr}\ \mathrm{COMMA}\ \textit{args}$$

Notes about Formal Languages

- Various methods exist to recognize and parse various kinds of formal languages
  - **Recursive-descent** parsing is popular because it's conceptually simple but has fallen out of favor because it's slow and inefficient
  - **LALR-1** (lookahead-1 left-right) parsing is used in most modern programming languages, but it's relatively complex
- Some grammars may be **ambiguous**, meaning that there are multiple ways to produce the same string (we must specify things like order of operations and associativity)
- There are various related classes of languages (context-free is one of them)
- Formal language theory is intimately related to natural language processing, linguistics, automata theory, and theory of computation
- Anyone sensing another lecture?

Semantic Analysis

- Once we've figured out the syntactic structure of our program, we still have more work to do before actually generating code
- Semantic analysis basically includes doing a bunch of things to work out the "meaning" of our code before we actually generate output, including:
  - **Type checking**: assigning every expression a type and ensuring that all types are correct and there are no type mismatches (this is done very differently in different languages)
  - Checking for multiple definitions of functions and variables
  - Checking that functions and variables can each be matched to a definition
- In general, semantic analysis adds information to the abstract syntax tree, creating an **attributed abstract syntax tree**

Type Checking

- There are multiple ways that type checking can work in programming languages
- **Static type checking** is type checking done at compile time
- **Dynamic type checking** is type checking done at runtime
- Static type checking is faster and safer (makes better guarantees), but it's less flexible and doesn't allow some useful features
- There are three major type systems
  - **Structural typing**, in which objects' types are defined by their actual structure and not their name
  - **Nominative typing**, in which objects' types are defined by explicit declaration
  - **Duck typing**, in which objects' types aren't checked but rather if they possess some functionality requisite in some scenario: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a

Optimization: definitions

Definition (Optimizing compiler)

A compiler that is built to minimize or maximize some attributes of a program's execution, generally through a sequence of **optimizing transformations**.

- Usually we're minimizing time, but also we sometimes want to minimize memory, program size, or power usage, especially in mobile devices
- Many optimization problems are NP-HARD or even undecidable, so in general, optimizing compilers use many heuristics and approximations and often don't come up with a near to ideal program
- Optimizations may be either **platform-dependent** or **platform-independent**

Types of optimization

Definition (Basic Block)

A code sequences that contains no jumps or branches besides the entrance into the sequence and exit from the sequence.

1. **Peephole.** Looks at a few instructions at a time, typically micro-optimizing small instruction sequences.
2. **Local.** Contained within one basic block.
3. **Loop.** Acts on a loop.
4. **Global.** Between multiple basic blocks in a single function.
5. **Interprocedural/whole-program.** Between multiple functions in a program.

Common techniques

1. **Strength reduction.** Complex computations are reduced to less “expensive”, but equivalent, computations in order to save computation time (or power consumption or whatever is being optimized).
2. **Avoid redundancy and eliminate dead stores/code.** Eliminate code that isn't used, variables that aren't used, and calculating the same thing multiple times.
3. **Elimination of jumps.** Jumps, loops, and function calls slow down programs significantly, so in some cases, for example, loops are unrolled at the cost of increasing binary program size.
4. **Fast path.** When there is a branch with two choices in a program, and one of them is much more common than the other, that one can automatically be assumed, and then "undone" if the condition turns out to be false.

Data-flow optimization

Noah Singer, George Klees

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

Data-flow optimization

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

2. **Constant propagation and constant folding**

```
int a = 8 * 3 + 2 / 7;
printf("%d\n", a+5);
```

Data-flow optimization

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

2. **Constant propagation and constant folding**

```
int a = 8 * 3 + 2 / 7;
printf("%d\n", a+5);
```

3. **Common subexpression elimination**

```
int a = (c * 3) + 47;
int b = (c * 3) % 2;
```

Data-flow optimization

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

2. **Constant propagation and constant folding**

```
int a = 8 * 3 + 2 / 7;
printf("%d\n", a+5);
```

3. **Common subexpression elimination**

```
int a = (c * 3) + 47;
int b = (c * 3) % 2;
```

4. **Dead store elimination**

Other optimizations

1. **Tail call elimination**

```
int fib(int product, int count, int max)
{
        if (count > max) return product;
        else return fib(product * count,
            count + 1, max);
}
```

2. **Strength reduction of multiplication and division**
3. **Function inlining**

Intermediate representation

Definition (Intermediate language)

A language used to describe code running on a theoretical, simple machine

- The AST is translated into an intermediate language, which the machine code is generated from
- Most intermediate languages (such as the common three-address code) have unlimited variables and can only do a single operation in one line

Intermediate representation

```
for (int i = 0; i < 8; i++)
        printf("%d\n", 2 * a / b - 7 * c);
```

```
r0 := 0
in_loop:
  r1 := 2 * a
  r2 := r1 / b
  r3 := 7 * c
  r4 := r3 - r2
  r5 := "%d\n"
  printf(r1, r5)
  r0 := r0 + 1
  jmp_nlt r0 8 done
done:
  exit
```

Registers, caches, and RAM

Definition (Memory)

A piece of hardware that stores binary data.

1. **Registers.** Internal CPU memory
   - Each register is a fixed size, and assigned a number
2. **Cache.** Multi-level buffer between CPU and RAM
3. **RAM.** External memory
   - Every byte (8-bits) assigned a **memory address**

Cache architecture

- Multiple cache levels (L1, L2, L3), with lower numbers having less memory, being faster, and shared among fewer CPUs
- A single cache consists of several **cache blocks**, holding data from RAM
- When the CPU accesses RAM, it first tries each level of the cache in ascending order (L1, L2, ...) before going to RAM
- Data not being found at a certain level is a **cache miss**, meaning the data must be retrieved from a higher level (performance penalty) and moved to the lower levels
- When a new block is read after a cache miss, a less-recently used block may be **evicted**

Memory access latency (2016)

1. Single clock cycle: <1 ns
2. L1 cache reference: 1 ns
3. L2 cache reference: 4 ns
4. Main memory reference: 100 ns
5. Read 1MB from disk: 1ms
6. Send packet to the Netherlands over network: 150 ms

Adapted from Colin Scott at UC Berkeley.
`http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

Locality of reference

Definition (Locality of reference)

Locality of reference occurs when memory accesses are correlated and close together.

1. **Temporal locality:** If a memory location is accessed, it will probably be accessed in the near-future
2. **Spatial locality:** If a memory location is accessed, nearby locations will probably be accessed in the near-future

- The memory hierarchy (registers, cache, RAM) is based on locality commonly used variables are kept in faster memory

Optimizing register use

- Registers are faster than cache or RAM, so all local variables should be in registers if possible
- If there are more variables than registers, the compiler must allocate the variables to either registers or the stack
    1. Construct a graph; variables are nodes, **interference edges** connect simultaneously-used variables, and **preference edges** connect one variable that's set to another
    2. With K registers available, **assign each node a color** such that: no nodes sharing an interference edge are the same color, and nodes sharing preference edges are the same color if possible
    3. Color of each variable is its register assignment

Optimizing the cache

- The idea of a cache is based on locality of reference, so code should take advantage of it
- Ensure spatial locality (and minimize cache misses) by keeping related data together
  - Keep code and data compact
  - Put data that will be accessed at similar times in the same cache block

The importance of spatial locality

- How would you multiply matrix A (m x n) by matrix B (p x q)?

```
for (int rA = 0; rA < n; rA++)
  for (int cB = 0; cB < p; cB++)
    for (int rB = 0; rB < q; rB++)
      C[rA][cB] += A[rA][rB] * B[rB][cB];
```

```
for (int rA = 0; rA < n; rA++)
  for (int rB = 0; rB < q; rB++)
    for (int cB = 0; cB < p; cB++)
      C[rA][cB] += A[rA][rB] * B[rB][cB];
```

Loop optimization

1. **Loop fission/distribution.** Split a loop into multiple sequential loops to improve locality of reference.
2. **Loop fusion/combination.** Combine multiple sequential loops (with the same iteration conditions) to reduce overhead.
3. **Loop interchange.** Switching an inner and outer loop in order to improve locality of reference.
4. **Loop-invariant code motion.** Move code that calculates some value that doesn't change to outside the loop.
5. **Loop unrolling.** Replace a loop that iterates some fixed N times with N copies of the loop body.