Final Exam Study Guide

CPSC 3520 - Brendan McGuire

Disclaimer: while I attempted to be as accurate as possible, I am a student just like you. If you see anything wrong or unclear here, please let me know and I will be sure to correct it!

Semantics

Programming languages are composed of syntax (the grammar of the symbols allowed), and semantics (what do those symbols actually mean). The format semantics of a programming language related the language syntax to meaning, usually a description of an underlying computation (*a + b* means add the value stored in a to the value stored in b).

Programming Language = Syntax + Semantics

To start, let's consider an example: the if statement in C. The syntax and an information description of the semantics are described below:

```
if (<condition>) <statement>
if (<condition>) <statement> else <statement>
```

In an if statement, the <condition> is first evaluated. If the condition has a nonzero value (it is not all zeros), then the statement directly following it is executed. If the condition's value is all zeros, then the statement after else is executed, provided it exists.

Semantic Equivalence

Two different program fragments may have exactly the same meaning. For example, in C, ternary expressions are just a more compact way of representing if statements. Thus, the following program fragments *have exactly the same meaning*

```
if (a > b) {
 c = true;
} else {
 c = false;
};
c = a > b ? true : false;
c = a > b;
```

The exact definition of semantic equivalence can depend on the way that you formalize your semantics. So, let's consider all of the ways we can more *formally* define semantics for a language.

> *Note: the way formal semantics are described can be quite abstract, and hard to reason about. I have done my best to simplify the definitions below, but you may need to read this section multiple times to fully understand the different methods of semantic formalism.*
>
> *The important thing to keep in mind for all of these definitions is that we are trying to define our language in terms of something else (in terms of another language, or a machine, or a series of mathematical constructs, etc). By relying on this existing semantic definition, we can give our language fragments a formal meaning.*

Bootstrapping (Semantics by Self-Definition)

One common way to establish semantics for a language is to define the interpretation of the language in the language itself. For example, the *C* compiler *gcc* is written in C. The TypeScript compiler is written in *TypeScript*, and a Lisp interpreter might be written in Lisp. This practice is known as **bootstrapping**.

While this may seem initially confusing, this is normally achieved by compiling the next version of the compiler based on the previous version of the language. An MVP compiler is written in a different language, until the language is sufficiently advanced to port the compiler to.

Translational Semantics
Semantics are often conveyed by translating a program fragment into another language (for example, bytecode or assembly language). This requires an actual or hypothetical machine to compile against. It is not necessarily mutually exclusive with *Bootstrapping*.

Operational Semantics
One way of defining semantics is by definition how the computation is performed for each language fragment. For example, in *Assembly*, there is a direct correspondence between each instruction and a computation on the machine. Also known as *intensional semantics*.

For this to work, an actual machine to do the computation must exist and be precisely defined. These machines tend to be simple.

Denotational Semantics
In denotational semantics, we formalize the meanings of programming languages by constructing mathematical objects that describe the meanings of expressions. We establish a mathematical mapping between a syntactic construct and a mathematical object. For example,

The semantic interpretation of the string 1101 is determined by mapping the ***semantic function*** *D, which outputs the number 1110. The string (15, 24) mapped through the semantic function might output a vector <15, 24>*

Axiomatic Semantics
In axiomatic semantics, the meaning of a syntactically correct program is formalized as a logical proposition or assertion. Program statements are based on assertions about logical relationships that remain the same. *The program reduces to a series of commands. We define the meaning of a command in a program by describing its effects on assertions about the program state.*

<u>*Algebraic Semantics*</u>
A form of axiomatic semantics, an algebraic specification of data and language constructs is developed. This is the underpinning for abstract data types, a common technique in Object Oriented programs.

Computing Impact

Software is increasingly important in the world. More and more aspects of our life rely on it, from physical products, to the services and institutions we interact with, to software that collects data about us. This is simultaneously concerning and exciting!

Software Responsibility

Unlike many other engineering disciplines, there is no certification for computer scientists or engineers. Additionally, many product liabilities laws do not apply to software products, due to the software's attached *End User License Agreement.* These contracts apply to you by using this software, and very often disclaim all liability of the software producer for any harm that results from use of the software. Such liability disclaimers have been upheld by the courts.

Safety & Functionality

In most cases, the limited liability of software is not a huge deal (*it doesn't really matter if Facebook malfunctions)*, but software is increasingly being used to influence real hardware devices that can have massive implications

- ➔ Software control of planes and cars
- ➔ Software that regulates the function of a pacemaker
- ➔ PLC code that controls expensive and precarious equipment
- ➔ A website that accidentally doxxed you to the world.

Environmental Issues

Computing devices are quite demanding of resources, energy, and time. Consider some of the ethical environmental issues surrounding computers:

- ➔ 2% of the world's energy goes to powering computers. How can we ensure that energy is most efficiently spent?
- ➔ Very often, hardware requires very specific and hard to come across natural minerals and materials. How can we ensure these materials are collected in a sustainable and ethical manner?
- ➔ How do we safely and sustainably get rid of electronics when they are through their usefulness?

How can we use computers to help the environment?

- ➔ Intelligently manage thermostats to save energy
- ➔ Simulate climate models to help us understand climate change
- ➔ Reduce paper usage

Computing Ethics

The application of moral principles to our field of computing. Some computing ethical issues to consider:

- ➔ Computers make it easier to violate intellectual property rights. To what extent should we minimize this, and how should we?
- ➔ What are the social impacts of computers/AI in our lives? How can we design systems to interact better with humans?
- ➔ How do we protect people from computer criminals? How do we protect our privacy from malicious actors?

Software Licenses

Unlike physical products, when you purchase or use software, you are agreeing to a license to use the software. The exact details and restrictions of this license may vary, but courts have held that software licenses are binding contracts.

Free Software
A large amount of software is licensed freely, in a copyleft fashion. Specifically, the General Public License grants the freedom to use the software for any purpose, to change it to suit your needs, to share it with others, and share the changes you make.

The Types of Free. Here, we are conflating a number of different definitions of free in the context of software. "Free-As-In-Beer" software is distributed without money needing to change hands. "Free-As-In-Speech" software has very few restrictions about what you can do with it. See the table below for examples of the different types of free software.

	Free As In Beer (Gratis)	**Not Free As In Beer**
Free As In Speech (Libre)	*Ex: Linux Kernel, gcc* Software is freely distributed, and you are not restricted in what you can do with it.	This is a small category, but often includes software that is open source and can be compiled, but where executable versions are sold.
Not Free As In Speech	*Ex: Facebook App* Software is freely distributed, but modifying it against the *Terms of Service*	*Ex: Photoshop* Software is a subscription service, and you are not permitted to modify it, redistribute it, or change it.

GNU Public License
The GPL is a copyleft license designed to thwart efforts to prevent one from writing and distributing free software. It has conditions that restrict:

- ➔ TIVOization: hardware that runs GPL software but prevents you from changing that software
- ➔ DMCA, EUCD: attempt to make it illegal to do certain things – like crack DRM
- ➔ Discriminatory patents such as: those attempted by MS trying to collect royalties for running free software on their platform.

Any software that is licensed with GPL may be copied and shared, ***so long as the copies are also GPL licensed.*** This is known as the GPL glom-on, and is intended to prevent someone from taking free code and making it private.

LGPL (Lesser GPL)
The same as GPL, but without the glom-on, usually intended for libraries. Does not force developers to adopt GPL just because they want to use the features of the library.

Event-Driven Programming

When dealing with user input and graphical user interfaces, the order of events is not deterministic. For example, you never know when the user is going to click a button. The traditional sequence events for programs looks like:

get input ➔ *compute* ➔ *output* ➔ *halt*

In this developer-driven control flow paradigm, the programmer knows the exact sequence of events, and user input is solicited from the program.

Important: Nowadays, lots of software has shifted paradigms, where instead of prompting for user input, the job of the program is to *react* to user actions.

Event Loop

In order to handle this paradigm shift, many event-driven programs rely on an event loop. This is an infinite loop that waits for and then dispatches reactions to specific events. When doing this, it is important to first determine what the graphical user interface looks like, and anticipate all possible events the user can perform on it.

We typically respond to an event by executing a piece of code that "handles" that event. Mapping an event to its handler is an important first step.

Parallel Programming

In recent years, single-core performance of our computers has stalled. While gains are still being made, they are relatively small. To improve performance, hardware manufacturers have additional parallel capabilities to the computers; often consumer CPUs can have 16 or 32 cores, which are all capable of independent and simultaneous computation.

In order to leverage this massive parallelism, we need to be able to split our programs up between these processors. This means *decomposing our algorithms into parallel applications.* In some applications, this may be easier, but in others it could be incredibly difficult.

Example*: Consider a large array of integers. Write a parallel algorithm that will quickly square all of the numbers.*

This problem is trivial to parallelize. Simply divide the array into n smaller subarrays, where n is the number of CPU cores you want to use, and have each core work independently on its small subproblem. Once all cores have completed their work, reassemble the array.

Oftentimes, problems that have large numbers of dependencies on previous data can be hard to parallelize easily

Example*: Write a parallel algorithm that prints the first 100000 number of the fibonacci sequence.*

It is not trivial to divide up this into multiple processors because each number depends on the previous two answers. There are mathematical formulas which can be used to compute the nth fibonacci number without relying on previous answers, but they are not immediately obvious.

One might hope that an *n* processor implementation of an algorithm will achieve the result in 1/nth the time required by a single processor, but because of these dependencies and communication required between processors, this is at best an upper bound.

Metrics for Parallelization Speedup

For a parallel implementation, *speedup* is the ratio of the processing time with a single processor over the processing time with n processors.

$$speedup = \frac{t_{single\ processor}}{t_{n\ processors}}$$

As previously mentioned, the upper bound for this speed up is n.

<u>Amdahl's Law</u>

One way to estimate speedup is using Amdahl's Law. This law bases speed up on the fraction of computations believed to be parallelizable as compared to the fraction of computations which must be performed sequentially.

$$speedup = \frac{s+p}{s+p/n} = \frac{1}{s+p/n}$$

Where:

s – the fraction of computations which are strictly sequential

p - the fraction of computations which are parallelizable (s + p = 1)

n – the number of processors over which p is distributed

MPI (Message Passing Interface)

There are several important mechanisms in parallel computing:

1. A means to to mark a code segment as PARALLEL or SERIAL (or some combination thereof)
2. A means to indicate a block should be processed in parallel
3. A means to allow synchronization or communication between processes, to resolve dependencies.

MPI, or Message Passing Interface, is the latter, and facilitates process communication and synchronization using a library of functions. This is mostly achieved by the transmission of data between a pair of processes (point to point communication).

<u>SIMD (Single Instruction, Multiple Data)</u>

One way to characterize parallel programming is through SIMD instructions. These will all perform the same operation on a large vector of data. MPI often extends this idea by SPMD (Single Program, Multiple Data). The program is distributed to each of the processes, and is executed one each.

<u>Process Rank</u>

Each process is identified by a specific process rank, an integer which distinguishes the process. This can be used to tailor the program to each process.

Example Program

Consider the following example program, which spawns a number of processes that all send a message back to the main process.

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc,char** argv){
  int p;/*Rank of process*/
  int n;/*Number Of Processes*/
  int source;/*Rank of sender*/
  int dest;/*Rank Receiver*/
  int tag=50;/*Tag for messages*/
  char message[100];//storageformessage

  MPI_Status status;/*Return status for receive*/
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&p);
  MPI_Comm_size(MPI_COMM_WORLD,&n);

  // In all of the child processes, send a message
  if(p != 0) {
    sprintf(message,"Hello World from process %d", p);
    dest=0;
    MPI_Send(message,strlen(message)+1,MPI_CHAR,dest, tag,MPI_COMM_WORLD);
  } else {

    // In the original process, receive messages from all the others
    for(source=1;source<n;source++){
      MPI_Recv(message,100,MPI_CHAR,source,tag,
      MPI_COMM_WORLD,&status);
      printf("%s\n",message);
    }

  }

  MPI_Finalize();
}
```