Programming

Kevin Sullivan

August 4, 2015

Contents

1 Preface

This book has been prepared for students taking Kevin Sullivan's Computer Science 1113 class, also known as CS1, at the University of Virginia, in the Fall Semester of 2015. The course is meant to give students with no previous exposure to computer science an accessible yet rigorous introduction to computational thinking, and to equip such students with the skills needed to use the fundamental tools used in software development practice around the world.

This course will focus on both fundamental and intellectually deep concepts and on developing practical performance skills. It will teach and illustrate concepts using two programming languages, one hosen for expressive clarity (a pure functional programming language), and one chosen practical utility and industrial relevance (Python). It will cover many examples, with an emphasis on examples relevant to data science.

1.1 Structure Based on Fundamental Data Types

In his famous book on Software Engineering, The Mythical Man-Month, the computer scientist, Fred Brooks, said, "Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious."

Designing good data representations is the foundation for efficient, reliable, and understandable programs. Code is simply instructions that define and manipulate data, so if the data representations in a program are a mess, then the code will be a mess. If, on the other hand, the data representations are crisp, clean, and appropriate, then the code will almost write itself.

The organizing principle of this book is that of fundamental abstract data types (ADT) of programming. These data types transcend individual programming languages and methods. They are among the most enduring of programming principles. Each of the first ten or so chapters of this book thus introduces a new fundamental ADT, including a data type defining the terms of the type, and a set of functions taking such terms as their parameters.

A broad range of foundational concepts of programming are then explored within this overall, data-centric structure. Concepts covered in this book include types and terms, function types, expressions and values, inductive definition, structural recursion and termination, typeclasses, polymorphic types, expresssions and their evaluation, operators, precedence, associativity, partial evaluation, higher-order functions, (co-inductive) processes, and abstraction from concrete representations, including the concept of representation invariants.

1.2 Programming Paradigms and Languages

Many of the foundational data types and principles of computing can be presented in their cleanest, cleared, and most general forms using not only a pure functional programming language, but one of the even more novel languages based on constructive logic and dependent types. On the other hand, these languages lack the libraries, economics, and other resources that have evolved around popular industrial languages, such as Java and Python.

This class will use both: a language with a dependent type system and optional enforcement of totality to teach the fundamentals of data, functions, expressions and their evaluation, and to a limited extent stateful programming and effects (i.e., type thoery). It will use a popular industrial imperative language (such as Python) to teach the practical applications of these fundamental concepts.

The anticipated cost of this approach is that learning two languages will be harder for students than learning one. The expected benefit is that students will develop a deeper understanding of the otherwise apparently somewhat arbitrary feature sets of our modern, industrial programming languages. The Virtual Machine that accompanies this book is configured to ease the task of switching between languages. For example, students will use

the same text editor for both languages.

Early introduction to functional programming is also expected to prepare students to use the most recent and advanced features of such industrial languages. Such features increasingly come from the realm of functional programming. Such features include lambda abstractions and higher-order functions.

Finally, the decision to teach students fundamentals of programming using a language based on dependent type theory is meant to prepare students for follow-on courses on formal specification of languages and properties of programs and the deductive verification of logical propositions about programs.

1.3 Learning by Coached Doing

This class is intended to be taught in a highly interactive manner that always proceeds from concrete to abstract. Students should always come to class. They should always bring their laptop computers. They should always follow along with the instructor as he or she develops the concrete examples through which the more abstract ideas are developed, illustrated, and made meaningful.

The understanding of abstract ideas arises most efficiently through generalization from concrete and meaningful examples. It is alright to briefly introduce the abstract concept that is meant to be taught, but the concept should then be developed through concrete examples, with the abstract concept being fully articulated once it is clear that the students have grasped the individual, meaningful examples.

1.4 Prerequisites

The course assumes that entering students have no prior exposure to programming or computer science. It is meant to provide such students an interesting and accessible path into the field. Success in computer science requires no special intellectual talents. It just requires that students work hard to understand the concepts and to develop the skills needed to use programming to creatively solve interesting and important provlems efficiently and reliably.

1.5 Objectives

By the end of this course, students should have a good understanding of computation and programming, and a solid foundation of knowledge and

skills for a major or a career in computer science, in software development, or any other field—from finance to medicine to natural or social science to entrepreneurship—that requires the development and use of software and computation.

1.6 Duration

The course is designed for a 14-week semester. A 10-week version can be taught in a quarter using an initial sequence book chapters. This introduction is meant to be read before the course starts. The rest of the book provides one chapter for each week of class. The book assumes that each week includes three hours in class an one hour of hands-on laboratory practice.

2 Introduction

This introductory chapter gives an overview of many, though not all, of the main concepts and themes of this course. It's not just an informal introduction, though. Rather, it develops some of the most important ideas and terminology in this course. You should study this chapter with some care, and ask lots of questions. Topics in this chapter, elaborated in the rest of the book, include the following:

- the nature of *data*;
- that data represent *mathematical objects*;
- that such objects have *mathematical properties*;
- that *interpretations* are required to give such objects real-world meanings;
- that functions *transform* data and, when used properly, real-world meanings;
- that functions can be extended to have *effects* in the real world;
- that functions can be expressed as data, which we call *programs*

Let's dive in and explore these ideas in a litle more detail.

2.1 Data

Data are physically recorded representations of mathematical objects. Such mathematical objects can be simple or extraordinarily complex. Simple examples include the natural numbers, 2 and 0, and the *truth values* of Boolean logic, *true* and *false*.

A slightly more complex example of a mathematical object that could be represented by data is a sequence of ordered pairs, where each pair has as its first element a sequence of letters, and as its second element, a 10-digit decimal numeral. Take a moment to write down an example of data of this type on a piece of paper. Can you think of examples where data of this type might be useful?

Complex mathematical objects are often structured assemblages of less complex objects. In the the preceding example, a sequence of pairs is made up of individual pairs. Each pair is made up of a sequence of letters and a 10-digit numeral. Each sequence of letters is made up of individual letters, and the numeral is a sequence of digits.

An even more complex example of a mathematical object that could be represented by data is that of a mathematical function. A particular example is that of the function, $+$, that maps any pair of natural numbers to their sum (e.g., $2 + 2$ to 4). The data representing such a function could, for instance be a precise textual description of a procedure to compute the result of applying $+$ to any pair of natural numbers. We could call such a description a *program*.

2.2 Representation

As we said to begin, data are *representations* of mathematical objects. In general, any given mathematical object can be represented in many ways. For example, the natural number, two (a mathematical object, for sure) can be represented by the decimal numeral, 2, by the binary numeral 10, or by the unary numeral, ||, (two slashes, as you might write when keeping a tally of events on a piece of paper).

We will sometimes call such representations *terms*. Data values are thus terms that represent mathematical objects. We will also often classify terms into *types*. For example, a type of unary terms for representing the natural numbers might include a term to represent 0, perhaps just the letter, Z, and then the additional terms $|Z$, to represent one, $||Z$, for two, $|||Z$ for three, and so forth. The type of binary terms for representing the natural numbers includes the numerals 0, for zero, and then, going up, $1" for one, 10 for

two, 11 for three, 100 for four, 101 for five, 110 for six, 111 for seven, 1000 for eight, 1001 for nine, and so forth.

Such terms can in turn be physically recorded in many ways: as ink on paper, marks in sand, as electrically charged or discharged states of capacitors, as patterns of pixels on a screen, or in many other ways. Modern digital computers use a variety of physical media to record data. They include the use of tiny capacitors (random access memory, or RAM, and flash memory), magnetic spots on metallic disks (hard drives), and small pits, made and read by lasers, in rotating plastic disks (DVD drives).

Details of the physical media are mostly irrelevant to this class. From this point on, think of data not as physical representations of mathematical objects, but just as mathematical objects abstracted from details of physical representations. For example, you may think of mathematical objects, such as the numeral 2, as data, no matter how they are physically recorded.

2.3 Properties

As mathematical objects, data, such as 2 and 0, lack specified real-world meanings, but they do have mathematical properties. For example, no one has yet said whether 2 and 0 in a particular context are meant to represent speeds, the number of apples in a basket, or a wager in a game of poker. Yet these data do have the property that $2 + 0 = 2$. This property derives from the definition of the mathematical objects that the data represent: here the natural numbers.

Indeed, the simple *proposition*, $2 + 0 = 2$, exhibits several properties of natural numbers. In particular, we can add pairs of such numbers using the $+$ function, and we can assert that two numbers are equal using $=$. In sum, data represent mathematical objects that in general come equipped with certain mathematical properties, including operations such as $+$ and $=$. What data don't natively come with are *interpretations*, the subject of the next part of this chapter.

2.4 Interpretations

What data on their own do not have are real-world meanings. Think about the natural number, 2, or 0, or the character {/em string}, "1X4NGP". Does 2 mean the number of boys in my family, the speed of a hockey puck, the number of apples in my basket, or the amount of a drug to administer in milligrams? Is that string a password for my Amazon.COM account, or a product code? Data themselves do not say.

Rather, we must associate real-world meanings with data by imposing what we call {/em interpretations} on data values. An interpretation is an explanation of the intended meaning of one or more data values, in a particular context. Clearly we will {/em interpret} the number, 2, as having different meanings in different contexts.

Interpretations also constrain the ways in which the mathematical properties of data can be used. If we use the natural number, 2, to represent the number of apples in one basket, and the number, 2 (again), to represent the number of oranges in another basket, then even though the natural number {/em data type} has a + function that could be used to compute that $2+2$ yeilds 4, the 4 does not have a valid interpretation, assuming that it makes no sense to add apples and oranges.

What we really see here is that, if we are to use such values and functions to represent things and functions in the real world, then we must define interpretations not only of given {/em values} of data types, but for their functions, as well. If in the fruit basket case, for example, we interpret the + function as summing up the number of {/em fruits} in the two baskets, then we can interpet the result, 4, as having a valid real-world meaning.

2.5 Functions

Functions are mathematical objects that define associations between data values of given types. For example, the + function, discussed above, associates the natural numbers, 2 and 0, with the natural number, 2, and the numbers, 2 and 2 with the number 4.

What we are usually concerned with in computer science is the ability to automatically {/em evaluate} the {/em application} of a {/em function} to one or more {/em parameter} data values to produce a {/em result} value. Here, for example, we want to evaluate the application of the function, +, to the parameters, 2 and 2, to produce the result, 4.

We can thus think of functions as representing {/em transformations} that convert data in one form (parameters) into data of another form (results). The real power of functions viewed in this way is that they can convert many values of given types into results. For example, + is not only capable of transforming 2 and 2 into 4, but it can transform any two natural numbers, x and y into a result, namely one that represents the arithmetic sum, $x + y.$ Functions thus {/em abstract} from specific values to generalized {/em parameters.}

We started off this chapter by saying that data represent mathematical objects. Now we've said that functions are mathematical objects, too.

This conjunction of ideas leads us to ask whether data can represent functions? Indeed they can. And this idea is at the very heart of the concept of programming!

We can create data objects that represent functions. We can even use function values as parameters to and as results of other functions! Once we have data values representing functions, we can then write {/em expressions} in which we {/em apply} functions to parameters, as in the expression, $2 + 2,$ representing the application of + to the values, 2 and 2.

As a {/em coup de grace,} computers are programmed to {/em evaluate} such expressions automatically—to automatically transform give data to produce mathematically valid, and often extremely useful, results. Learning to define your own data and function types and values, to imbue them with meaningful interpretations, and to use the resulting capabilities to automatically compute meaningful results is the very heart of this class, and the essence of the programming activity.

2.6 Effects

The ability of computer to compute the results of applying functions to data is incredible. To be useful, however, computers must do more than {/em compute}; they must also {/em interact} with the real world around them. They must be able to sense the world, to obtain data about it, on which to compute. And they must be able to have {/em effects} on the world, so as to perform useful actions.

Computers are thus equipped with the capacity both to compute, which is purely mathematical activity that can be carried out within the confines of a computing machine, and to interact: with us, with other computers, and with physical devices. Devices include {/em sensors}, which convert physical phenomena, such as sound, light, and pressure, into data; And they include actuators, which convert data into actions in the physical world (such as driving speakers that play music into your ears, based on numeric data stored in your phone).

In many ways, it's not an overstatement to assert that programming is all about data, including functions, and {/em effects}, which is the term we will use to encompass all interactions with the physical world. Effects include changes in the states of devices attached to computers, from keyboards and mice (kinds of sensors), and screens (a kind of actuator), to networks, other computers, and more interesting physical mechanisms, such as the actuators that control robot motions or the steering and safety functions of self-driving cars.

In this class, we will spend several weeks almost entirely in the realm of pure functions, except that we will use effects to provide parameter values to functions and to see the results of computations on our screens. Later on in this course, we will look more deeply at the concept of effects, including particular several forms of what we call I/O (pronounced eye-oh'), or input-output actions.

You will learn about obtaining data on which to compute by using input obtained from the keyboard, mouse, files on your hard drive, and from network-accessible resources, such as web sites and online databases. You will learn about how to produce output to your screen, to files on your hard drive, and to send outputs to external sources such as web services (where they will be used as inputs to remote computations).

2.7 Languages

2.8 This Book

3 Tools

This chapter introduces some of the most widely used software development tools in the world. In this course, you will learn to use these tools. Practicing developers around the world use these very tools on a daily basis. The tools you will learn about here are

- the *Linux* operating system;
- the *emacs* text editor ;
- several *compilers*;
- and the *git* and *GitHub* version control tools.

Don't worry that you don't yet know what all these words mean. That's what you're here to learn.

3.1 What about IDEs?

You will find that many other introductory classes *Integrated Development Environments* (IDEs) for programming. An IDE combines a compiler, a text editor, and some other tools, into a single program. An IDE is something like Microsoft Word for writing programs. It does a lot for you, but it also locks you into one particular environment.

IDEs have several advantages. First, it's easier to get started in programming with a simple IDE. Another is that IDEs *integrate* tools, particularly text editors and compilers, which means that IDEs make them work together seamlessly and automatically, so that you don't have to run and coordinated them manually.

In our approach, you will use text editing program, *emacs*, to edit a program. You will then issue a separate command to the Linux operating system to compile your program, transforming it into a form in which you can run it. Finally you will issue another command to Linux to run your program.

These are the tasks that an IDE handles for you at the click of a button. On the other hand, IDEs hide from you the details of what is actually happening as you develop software. Seeing what's really going on at the Linux command line will give you a deeper understanding of how software is actually produced.

Fortunately, *emacs* can be configured (and on your VM is configured) to provide most of the functionality of an IDE.

There are also several downsides to using a traditional IDE. One is that IDEs hide some of the important details about what's really happening when programs are compiled, run, and debugged. Another is that you can end up stuck in a very limited environment. This is particular true with the simple IDEs that are often used for teaching. Finally, most practicing programmers do not use IDEs.

The advantages of our approach is that once you learn to use Linux, emacs, compilers and debuggers, and git and GitHub, the world is your oyster. You're "set for life." These really are the tools that most practicing programmers use. If you go into a tech company, you are far more likely to see people using these tools than using an IDE. That's just the way it is. So the decision for this class is that we will pay the up-front cost of learning the standard tools, and we (and you) will reap the benefits in the days, weeks, months, years, and decades to come.

Such tools really do matter enormously in software development. Once you learn them, they give you tremendous leverage and efficiency. It always takes some time to learn to use new tools, but doing so is a routine part of the craft of coding, and the benefits that you will reap over time will dwarf the initial investment in learning them.

We will take a week to learn the tools that you will need for this course. By the end of a week, you'll be familiar enough with them to use them to good effect. After a few weeks, you'll no longer have to think to use them effectively. Like a hammer is an extension of one's arm that amplifies

its strength, software tools are extensions and amplifiers of the software developer's mind. They are simple things, in the end, but profoundly useful.

Configuring an operating system with the necessary tools is a chore. You will want to learn how to do it on your own eventually. We've done that work for you for this class. We will provide you with an Ubuntu virtual machine with all the required software installed and configured. Running the virtual machine on your own computer will in essence give you an entire Linux operating system (like a whole new computer!) with all the necessary tools installed and ready to go.

3.2 Virtual Machines: VirtualBox

3.3 Operating System: Linux

We will use the Ubuntu version of Linux, specifically version 14.04 Ubuntu. Linux is the operating system that is most commonly used by software developers, and Ubuntu is a very popular version of Linux. Unlike Microsoft's Windows or Apple's OS X, it's free and it runs on a very broad range of hardware platforms, including all modern PCs. It's also a vastly better environment in which to develop software than is Windows. The Mac OS X operating system is actually a close cousin of Linux. You can access the Linux-like features of OS X by using the terminal application on your Mac.

3.3.1 Terminals and shells

3.3.2 Running programs

3.3.3 The Linux file system

3.3.4 Operations on files

3.3.5 File names and types

3.4 Text Editors: Emacs

Now that you know how to move, rename, print the contents of files, and delete files, it's time to learn how to create files with useful content. Almost all of the files you'll create when using Linux will be plain text files. To create them, in this class, you will use a program called *emacs*.

Emacs is a text editor. It's analogous to Microsoft Word, but it's used to edit plain text, as opposed to typeset, documents. Programs are plain text documents, so emacs is a plausible choice for editing programs.

Emacs is also a *programmable* editor. It can be programmed to treat certain kinds of text files in special ways. For example, it can be programmed to provide special functions for editing programs in given languages, such as C++, Java, Idris, Haskell, or Python.

Emacs is actually so programmable, and has been programmed to do so many things, that some people thing of it as almost like an operating system itself. Among other things, it has for working with git, for compiling and running programs, and for many other tasks—all from within the editor. As you will learn over the course of this class, emacs is the equal to, and in many ways surpasses, most IDEs, in the ways it can help you as a programmer.

Emacs has been in use for decades. An incredibly rich library of custom-programmed extensions is available for emacs. We have already loaded several for you, including extensions for editing Python and Idris programs, for interacting with git, and for creating literate programs using *Org*. Literate programs combine text, graphics, and code. They're like living books. Emacs really is quite incredible.

Emacs is one of several popular, programmable, text editors. Others include vim and sublime text. Emacs is the choice that works best for this class. It is a popular text editor among data scientists. We have programmed it for your use in this class. It is free, and it runs on almost any computer you're ever likely to use. You can install it on Windows, Macs, and, of course, Linux.

Emacs is an example of a *software tool*. The concept of software tools is important. Tools enable us to perform software-related tasks in a far more efficient and reliable manner than would be possible without them. Learning any given tool requires an investment of time and effort. Yet just as with tools for carpentry or painting, such investments pay huge dividends over time. They don't only improve your efficiency and reliability; they ultimately make it possible for you to accomplish things that would otherwise not be feasible.

There is nothing complicated about the tools you will use in this class. That said, they can be confusing at first, and they do take some time to learn. Mostly you just need to memorize commands that won't seem natural at first. Practice is essential. Stick with it practice, ask questions, read documentation; post quick reference cards next to your desk. Soon the tools you are using will feel like powerful and natural extensions of your own hands and mind.

There are many emacs tutorials online, both on web sites and on such media as Youtube. In fact, emacs has an interactive tutorial built in. Your first learning task with emacs is to run through that tutorial. It will take you perhaps half an hour.

3.5 Compilers:

Compilers are tools that convert programs in the form of text files into programs in a form (called object code) that a computer can run directly. Any given compiler is designed to take programs written in one particular programming language. Thus Java compilers translate text files containing Java code into object code; Idris compilers translate text files containing Idris code into object code; and Python compilers translate text files containing Python code into object code.

We have installed compilers for numerous languages on the Virtual Machine we have provided you. The languages include Java, Python, Idris, Haskell, and Javascript (NodeJS). We have also configured emacs to support editing of program text files written in these languages. Indeed, you can not only edit such programs from within emacs, but you can compile and run them as well. We will coach you through the use of a few cryptic emacs commands to carry out these tasks.

3.6 Version Control: Git and GitHub

Finally, Git is what is call a distributed version control system. It serves two fundamental purposes. First, it allows you to keep track of multiple versions of documents (including programs) that you are editing. Second, it allows you to share changes to documents with collaborators. Because modern software development is almost always a "team sport," the use of a distributed version control system to coordinate changes to documents that are being developed jointly by multiple people is very important. Git is also free, and it also runs on a broad range of platforms.

3.7 Exercises

3.7.1 VM

3.7.2 Linux

3.7.3 Emacs

1. Tutorial Work you way all the way through the emacs tutorial in emacs.

2. Research Search Youtube for a good introductory tutorial on emacs. Watch it. Then use a search engine to find a good web site tutorial on emacs for beginners. Read it carefully.

3. Practice Finally, based on what you've learned, use emacs to write and revise a short essay, in a plain-text file called essay.txt, that expresses your hopes, fears, and goals for your first class in computer science. Your essay should be well written. Use this exercise to practice using as many emacs commands as you can. Be sure that by the time you are done, you can quickly create, edit, save, and open plain text files in emacs.

4 Logic and Language

In this chapter we will fully develop a language and interpreter for writing and evaluating expressions in Boolean algebra. If this sounds a little scary, don't worry, it's not.

4.1 Boolean Algebra

4.1.1 modules

```
module bool
```

4.1.2 types and values

1. true and false

   Our bool type has two of what we call "value constructors" (or just "constructors"). The constructor, "true" gives us one way to "build a value" of type bool. That value is the term "true" all by itself. The constructor, "false," gives us the other way to build a value of type bool; and that value is again just the term "false" all by itself. The type bool thus has, and is said to be "inhabited by" the two values (or equivalently "terms") "true" and "false."

   We can thus say that "a value of type bool is either true OR it's false. " We call such a type a "sum" type. Furthermore, in Idris and related languages, values produced by different constructors are always understood to be different: never just different names for the same thing. (Technically, we say that constructors are "injective.") Moreover, in Idris and similar languages, the values produced by the constructors of a type are the **only** inhabitances of that type. No values other than those produced by constructors inhabit can creep in to inhabit a given type.

2. the Boolean data type

   We define our own Boolean abstract data type type, with the date type name in lower case to avoid conflicts with Idris's standard Bool type.

   ```
   data bool = true | false
   ```

4.1.3 operations

1. unary There are four total functions that transform one bool to another. We sometimes call functions that take one argument "unary operations." Here we would say that we are definiting "unary operations on Booleans." What we mean is that we're defining functions that take one Boolean value as an argument (and that return some value). Here are the four functions.

   The identity function (on bool) reduces to argument

   (a) *id*

   ```
   id_bool: bool -> bool
   id_bool true = true
   id_bool false = false
   ```

   We can also write it more concisely.

   ```
   id_bool': bool -> bool
   id_bool' b = b
   ```

   And here's another way, where we name the argument up front

   ```
   id_bool'': (b: bool) -> bool
   id_bool'' b = b
   ```

   (b) *not* Here;s the function that returns the value that is not its argument.

   ```
   negb: bool -> bool
   negb true = false
   negb false = true
   ```

   (c) $constant_{true}$
   Here's the function that ignores its argument and always reduces to true.

   ```
   const_true: bool -> bool
   const_true b = true
   ```

(d) *constant*$_{false}$ Here's the function that ignores its argument and always reduces to false

```
const_false: bool -> bool
const_false b = true
```

2. binary

   (a) *and*
   Now we come to the binary operators.

   ```
   andb: bool -> bool -> bool
   andb true true = true
   andb true false = false
   andb false true = false
   andb false false = false
   ```

   (b) *or*

   ```
   orb: bool -> bool -> bool
   orb true true = true
   orb true false = true
   orb false true = true
   orb false false = false
   ```

   (c) *xor*

   (d) *nand*

   (e) *equality*

3. ternary: if-then-else

   If then else (ternary operator).

   ```
   ite: bool -> bool -> bool -> bool
   ite true tb fb = tb
   ite false tb fb = fb
   ```

4. type conversion

   ```
   boolToBool : bool -> Bool
   boolToBool false = False
   boolToBool true = True
   ```

4.2 Boolean Expressions and Evaluation

- using another module
- structure of a language
- abstract syntax
- literal expressions
- recursive expressions
- semantics
- recursive functions
- concrete syntax
- operator expressions
- variable expressions
- prefix, infix, postfix
- associativity, precedence

```
module boolexp
import public bool
-- abstract syntax

data bExp =
  litExp bool |
  notExp bExp |
  andExp bExp bExp |
  orExp bExp bExp |
  iteExp bExp bExp bExp

-- semantics

eval: bExp -> bool
eval (litExp b) = b
eval (notExp e) = negb (eval e)
eval (andExp e1 e2) = andb (eval e1) (eval e2)
eval (orExp e1 e2) = orb (eval e1) (eval e2)
```

```
-- concrete syntax

infixl 4 &&, ||

(&&): bExp -> bExp -> bExp
(&&) e1 e2 = andExp e1 e2
(||): bExp -> bExp -> bExp
(||) e1 e2 = orExp e1 e2

syntax True = litExp true
syntax False = litExp false
syntax IF [c] THEN [t] ELSE [f] = iteExp c t f

-- it'd be nice if this worked, but it doesn't (fully)
-- syntax "!" [b] =  notExp b
-- So we'' use this instead
syntax NOT [b] = notExp b
```

4.3 Real-World Applications

4.3.1 interpretations

4.3.2 satisfiability (and predicate logic)

1. validity
2. satisfiability
3. unsatisfiability

4.4 Exercises

5 Arithmetic

5.1 Natural Number

5.2 Arithmetic Operations

5.3 Expressions and Evaluation

5.4 Lab: Arithmetic Expressions

6 Geometry

6.1 Natural numbers and Peano Arithmetic

6.2 A library of arithmetic operations

6.3 An arithmetic expression language and interpreter

6.4 Lab: Arithmetic Expressions

7 Sequences

7.1 Polymorphic Pairs and Tuples

7.2 Polymorphic Lists

7.2.1 data type

7.2.2 operations

1. length
2. append
3. etc