1. Course Information

Staff

Faculty

Butler Lampson	32-G924	425-703-5925	blampson@microsoft.com
Daniel Jackson	32-G704	8-8471	dnj@mit.edu

Teaching Assistant

David Shin	dshin@mit.edu

Course Secretary

Maria Rebelo	32-G715	3-5895	mr@csail.mit.edu

Office Hours

Messrs. Lampson and Jackson will arrange individual appointments. David Shin will hold scheduled office hours in the 7th floor lounge of the Gates tower in building, Monday 4-6. In addition to holding regularly scheduled office hours, he will also be available by appointment.

Lectures and handouts

Lectures are held on Monday and Wednesday from 1:00 to 2:30PM in room 32-155. Messrs. Lampson and Jackson will split the lectures. The tentative schedule is at the end of this handout.

The source material for this course is an extensive set of handouts. There are about 400 pages of topic handouts that take the place of a textbook; you will need to study them to do well in the course. Since we don't want to simply repeat the written handouts in class, we will hand out the material for each lecture one week in advance. *We expect you to read the day's handouts before the class and come prepared to ask questions, discuss the material, or follow extensions of it or different ways of approaching the topic.*

Seven research papers supplement the topic handouts. In addition there are 5 problem sets, and the project described below. Solutions for each problem set will be available shortly after the due date.

There is a course Web page, at web.mit.edu/6.826/www. Last year's handouts can be found from this page. Current handouts will be placed on the Web as they are produced.

Current handouts will generally be available in lecture. If you miss any in lecture, you can obtain them afterwards from the course secretary. She keeps them in a file cabinet outside her office.

Problem sets

There is a problem set approximately once a week for the first half of the course. Problem sets are handed out on Wednesdays and are due by noon the following Wednesday in the tray on the

course secretary's desk. They normally cover the material discussed in class during the week they are handed out. Delayed submission of the solutions will be penalized, and no solutions will be accepted after Thursday 5:00PM.

Students in the class will be asked to help grade the problem sets. Each week a team of students will work with the TA to grade the week's problems. This takes about 3-4 hours. Each student will probably only have to do it once during the term.

We will try to return the graded problem sets, with solutions, within a week after their due date.

Policy on collaboration

We encourage discussion of the issues in the lectures, readings, and problem sets. However, if you collaborate on problem sets, you must tell us who your collaborators are. And in any case, you must write up all solutions on your own.

Project

During the last half of the course there is a project in which students will work in groups of three or so to apply the methods of the course to their own research projects. Each group will pick a real system, preferably one that some member of the group is actually working on but possibly one from a published paper or from someone else's research, and write:

> A specification for it.
>
> High-level code that captures the novel or tricky aspects of the actual implementation.
>
> The abstraction function and key invariants for the correctness of the code. This is not optional; if you can't write these things down, you don't understand what you are doing.

Depending on the difficulty of the specification and code, the group may also write a correctness proof for the code.

Projects may range in style from fairly formal, like handout 18 on consensus, in which the 'real system' is a simple one, to fairly informal (at least by the standards of this course), like the section on copying file systems in handout 7. These two handouts, along with the ones on naming, sequential transactions, concurrent transactions, and caching, are examples of the appropriate size and possible styles of a project.

The result of the project should be a write-up, in the style of one of these handouts. During the last two weeks of the course, each group will give a 25-minute presentation of its results. We have allocated four class periods for these presentations, which means that there will be twelve or fewer groups.

The projects will have five milestones. The purpose of these milestones is not to assign grades, but to make it possible for the instructors to keep track of how the projects are going and give everyone the best possible chance of a successful project

1. We will form the groups around March 2, to give most of the people that will drop the course a chance to do so.
2. Each group will write up a 2-3 page project proposal, present it to one of the instructors around spring break, and get feedback about how appropriate the project is and suggestions

on how to carry it out. Any project that seems to be seriously off the rails will have a second proposal meeting a week later.

3. Each group will submit a 5-10 page interim report in the middle of the project period.
4. Each group will give a presentation to the class during the last two weeks of classes.
5. Each group will submit a final report, which is due on Friday, May 14, the last day allowed by MIT regulations. Of course you are free to submit it early.

Half the groups will be 'early' ones; the other half will be 'late' ones that give their presentations one week later. The due dates of proposals and interim reports will be spread out over two weeks in the same way. See the schedule later in this handout for precise dates.

Grades

There are no exams. Grades are based 30% on the problem sets, 50% on the project, and 20% on class participation and quality and promptness of grading.

Course mailing list

A mailing list for course announcements—6.826-students@mit.edu—has been set up to include all students and the TA. If you do not receive any email from this mailing list within the first week, check with the TA. Another mailing list, 6.826-staff@mit.edu, sends email to the entire 6.826 staff.

Course Schedule

Date	No	By	HO	Topic	PS out	PS due
Wed., Feb. 8	1	L		**Overview**. The Spec language. State machine semantics. Examples of specifications and code.	1	
			1	Course information		
			2	Background		
			3	Introduction to Spec		
			4	Spec reference manual		
			5	Examples of specs and code		
Mon., Feb. 13	2	J		**Spec and code** for sequential programs. Correctness notions and proofs. Proof methods: abstraction functions and invariants.		
			6	Abstraction functions		
Wed., Feb. 15	3	L		**File systems 1**: Disks, simple sequential file system, caching, logs for crash recovery.	2	1
			7	Disks and file systems		
Tues., Feb. 21	4	L		**File systems 2**: Copying file system.		
Wed., Feb.22	5	L		**Proof methods**: History and prophecy variables; abstraction relations.	3	2
			8	History variables		
Mon., Feb. 27	6	S		**Semantics and proofs**: Formal sequential semantics of Spec.		
			9	Atomic semantics of Spec		
Wed., Mar. 1	7	J		**Naming**: Specs, variations, and examples of hierarchical naming.	Form groups	
			12	Naming		
			13	Paper: David Gifford et al, Semantic file systems, *Proc.13th ACM Symposium on Operating System Principles*, October 1991, pp 16-25.		
Mon., Mar. 6	8	L		**Performance**: How to get it, how to analyze it.	4	3
			10	Performance		
			11	Paper: Michael Schroeder and Michael Burrows, Performance of Firefly RPC, *ACM Transactions on Computer Systems* **8**, 1, February 1990, pp 1-17.		
Wed., Mar. 8	9	J		**Concurrency 1**: Practical concurrency, easy and hard. Easy concurrency using locks and condition variables. Problems with it: scheduling, deadlock.	5	4
			14	Practical concurrency		
			15	Concurrent disks		
			16	Paper: Andrew Birrell, An Introduction to Programming with C# Threads, Microsoft Research Technical Report MSR-TR-2005-68, May 2005.		

Date	No	By	HO	Topic	PS out	PS due
Mon., Mar. 13	10	S		**Concurrency 2:** Concurrency in Spec: threads and non-atomic semantics. Big atomic actions. Safety and liveness. Examples of concurrency.		
			17	Formal concurrency		
Wed., Mar. 15	11	S		**Concurrency 3:** Proving correctness of concurrent programs: assertional proofs, model checking		5
Mon., Mar. 20	12	L		**Distributed consensus 1**. Paxos algorithm for asynchronous consensus in the presence of faults.		
			18	Consensus		
Wed., Mar. 22	13	L		**Distributed consensus** 2.	Early proposals	
Mar. 27-31				Spring Break		
Mon., Apr. 3	14	J		**Sequential transactions** with caching.		
			19	Sequential transactions		
Wed., Apr. 5	15	J		**Concurrent transactions**: Specs for serializability. Ways to code the specs.	Late proposals	
			20	Concurrent transactions		
Mon., Apr. 10	16	J		**Distributed transactions**: Commit as a consensus problem. Two-phase commit. Optimizations.		
			27	Distributed transactions		
Wed., Apr. 12	17	L		**Introduction to distributed systems**: Characteristics of distributed systems. Physical, data link, and network layers. Design principles. **Networks 1**: Links. Point-to-point and broadcast networks.		
			21	Distributed systems		
			22	Paper: Michael Schroeder et al, Autonet: A high-speed, self-configuring local area network, *IEEE Journal on Selected Areas in Communications* **9**, 8, October 1991, pp 1318-1335.		
			23	Networks: Links and switches		
Mon., Apr. 17				Patriot's Day, no class		
Wed., Apr. 19	18	L		**Networks 2**: Links cont'd: Ethernet. Token Rings. Switches. Coding switches. Routing. Learning topologies and establishing routes.	Early interim reports	
Mon., Apr. 24	19	J		**Networks 3**: Network objects and remote procedure call (RPC).		
			24	Network objects		

Date	No	By	HO	Topic	PS out	PS due
			25	Paper: Andrew Birrell et al., Network objects, *Proc.14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993.		
Wed., Apr. 26	20	L		**Networks 4**: Reliable messages. 3-way handshake and clock code. TCP as a form of reliable messages.		Late interim reports
			26	Paper: Butler Lampson, Reliable messages and connection establishment. In *Distributed Systems*, ed. S. Mullender, Addison-Wesley, 1993, pp 251-281.		
Mon., May 1	21	J		**Replication and availability**: Coding replicated state machines using consensus. Applications to replicated storage.		
			28	Replication		
			29	Paper: Jim Gray and Andreas Reuter, Fault tolerance, in *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, pp 93-156.		
Wed., May 3	22	J		**Caching**: Maintaining coherent memory. Broadcast (snoopy) and directory protocols. Examples: multiprocessors, distributed shared memory, distributed file systems.		
			30	Concurrent caching		
Mon., May 8	23			**Early project presentations**		
Wed., May 10	24			**Early project presentations**		
Mon., May 15	25			**Late project presentations**		
Wed., May 17	26			**Late project presentations**		
Fri., May 19				**Final reports due**		
May 22-26				Finals week. There is no final for 6.826.		

2. Overview and Background

This is a course for computer system designers and builders, and for people who want to really understand how systems work, especially concurrent, distributed, and fault-tolerant systems.

The course teaches you
- how to write precise specifications for any kind of computer system,
- what it means for code to satisfy a specification, and
- how to prove that it does.

It also shows you how to use the same methods less formally, and gives you some suggestions for deciding how much formality is appropriate (less formality means less work, and often a more understandable spec, but also more chance to overlook an important detail).

The course also teaches you a lot about the topics in computer systems that we think are the most important: persistent storage, concurrency, naming, networks, distributed systems, transactions, fault tolerance, and caching. The emphasis is on
- careful specifications of subtle and sometimes complicated things,
- the important ideas behind good code, and
- how to understand what makes them actually work.

We spend most of our time on specific topics, but we use the general techniques throughout. We emphasize the ideas that different kinds of computer system have in common, even when they have different names.

The course uses a formal language called Spec for writing specs and code; you can think of it as a very high level programming language. There is a good deal of written introductory material on Spec (explanations and finger exercises) as well as a reference manual and a formal semantics. We introduce Spec ideas in class as we use them, but we do not devote class time to teaching Spec per se; we expect you to learn it on your own from the handouts. The one to concentrate on is handout 3, which has an informal introduction to the main features and lots of examples. Section 9 of handout 4, the reference manual, should also be useful. The rest of the reference manual is for reference, not for learning. Don't overlook the one page summary at the end of handout 3.

Because we write specs and do proofs, you need to know something about logic. Since many people don't, there is a concise treatment of the logic you will need at the end of this handout.

This is not a course in computer architecture, networks, operating systems, or databases. We will not talk in detail about how to code pipelines, memory interconnects, multiprocessors, routers, data link protocols, network management, virtual memory, scheduling, resource allocation, SQL, relational integrity, or TP monitors, although we will deal with many of the ideas that underlie these mechanisms.

Topics

General

Specifications as state machines.
The Spec language for describing state machines (writing specs and code).
What it means to implement a spec.

Using abstraction functions and invariants to prove that a program implements a spec.
What it means to have a crash.
What every system builder needs to know about performance.

Specific

Disks and file systems.
Practical concurrency using mutexes (locks) and condition variables; deadlock.
Hard concurrency (without locking): models, specs, proofs, and examples.
Transactions: simple, cached, concurrent, distributed.
Naming: principles, specs, and examples.
Distributed systems: communication, fault-tolerance, and autonomy.
Networking: links, switches, reliable messages and connections.
Remote procedure call and network objects.
Fault-tolerance, availability, consensus and replication.
Caching and distributed shared memory.

Previous editions of the course have also covered security (authentication, authorization, encryption, trust) and system management, but this year we are omitting these topics in order to spend more time on concurrency and semantics and to leave room for project presentations.

Prerequisites

There are no formal prerequisites for the course. However, we assume some knowledge both of computer systems and of mathematics. If you have taken 6.033 and 6.042, you should be in good shape. If you are missing some of this knowledge you can pick it up as we go, but if you are missing a lot of it you can expect to have serious trouble. It's also important to have a certain amount of maturity: enough experience with systems and mathematics to feel comfortable with the basic notions and to have some reliable intuition.

If you know the meaning of the following words, you have the necessary background. If a lot of them are unfamiliar, this course is probably not for you.

Systems

Cache, virtual memory, page table, pipeline
Process, scheduler, address space, priority
Thread, mutual exclusion (locking), semaphore, producer-consumer, deadlock
Transaction, commit, availability, relational data base, query, join
File system, directory, path name, striping, RAID
LAN, switch, routing, connection, flow control, congestion
Capability, access control list, principal (subject)

If you have not already studied Lampson's paper on hints for system design, you should do so as background for this course. It is Butler Lampson, Hints for computer system design, *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, October 1983, pp 33-48. There is a pointer to it on the course Web page.

Programming

Invariant, precondition, weakest precondition, fixed point
Procedure, recursion, stack

Data type, sub-type, type-checking, abstraction, representation
Object, method, inheritance
Data structures: list, hash table, binary search, B-tree, graph

Mathematics

Function, relation, set, transitive closure
Logic: proof, induction, de Morgan's laws, implication, predicate, quantifier
Probability: independent events, sampling, Poisson distribution
State machine, context-free grammar
Computational complexity, unsolvable problem

If you haven't been exposed to formal logic, you should study the summary at the end of this handout.

References

These are places to look when you want more information about some topic covered or alluded to in the course, or when you want to follow current research. You might also wish to consult Prof. Saltzer's bibliography for 6.033, which you can find on the course web page.

Books

Some of these are fat books better suited for reference than for reading cover to cover, especially Cormen, Leiserson, and Rivest, Jain, Mullender, Hennessy and Patterson, and Gray and Reuter. But the last two are pretty easy to read in spite of their encyclopedic character.

Specification: Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley, 2002. TLA+ is superficially quite different from Spec, but the same underneath. Lamport's approach is somewhat more mathematical than ours, but in the same spirit. You can find this book at http://research.microsoft.com/users/lamport/tla/book.html.

Systems programming: Greg Nelson, ed., *Systems Programming with Modula-3*, Prentice-Hall, 1991. Describes the language, which has all the useful features of C++ but is much simpler and less error-prone, and also shows how to use it for concurrency (a version of chapter 4 is a handout in this course), an efficiently customizable I/O streams package, and a window system.

Performance: Jon Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982. Short, concrete, and practical. Raj Jain, *The Art of Computer Systems Performance Analysis*, Wiley, 1991. Tells you much more than you need to know about this subject, but does have a lot of realistic examples.

Algorithms and data structures: Robert Sedgwick, *Algorithms*, Addison-Wesley, 1983. Short, and usually tells you what you need to know. Tom Cormen, Charles Leiserson, and Ron Rivest, *Introduction to Algorithms*, McGraw-Hill, 1989. Comprehensive, and sometimes valuable for that reason, but usually tells you a lot more than you need to know.

Distributed algorithms: Nancy Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996. The bible for distributed algorithms. Comprehensive, but a much more formal treatment than in this course. The topic is algorithms, not systems.

Computer architecture: John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann, 1995. The bible for computer architecture. The second edition has lots of interesting new material, especially on multiprocessor memory systems and interconnection networks. There's also a good appendix on computer arithmetic; it's useful to know where to find this information, though it has nothing to do with this course.

Transactions, data bases, and fault-tolerance: Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993. The bible for transaction processing, with much good material on data bases as well; it includes a lot of practical information that doesn't appear elsewhere in the literature.

Networks: Radia Perlman, *Interconnections: Bridges and Routers*, Addison-Wesley, 1992. Not exactly the bible for networking, but tells you nearly everything you might want to know about how packets are actually switched in computer networks.

Distributed systems: Sape Mullender, ed., *Distributed Systems*, 2nd ed., Addison-Wesley, 1993. A compendium by many authors that covers the field fairly well. Some chapters are much more theoretical than this course. Chapters 10 and 11 are handouts in this course. Chapters 1, 2, 8, and 12 are also recommended. Chapters 16 and 17 are the best you can do to learn about real-time computing; unfortunately, that is not saying much.

User interfaces: Alan Cooper, *About Face*, IDG Books, 1995. Principles, lots of examples, and opinionated advice, much of it good, from the original designer of Visual Basic.

Journals

You can find all of these in the CSAIL reading room in 32-G882. The cryptic strings in brackets are call numbers there. You can also find the ACM publications in the ACM digital library at www.acm.org.

For the current literature, the best sources are the proceedings of the following conferences. 'Sig' is short for "Special Interest Group", a subdivision of the ACM that deals with one field of computing. The relevant ones for systems are SigArch for computer architecture, SigPlan for programming languages, SigOps for operating systems, SigComm for communications, SigMod for data bases, and SigMetrics for performance measurement and analysis.

Symposium on Operating Systems Principles (SOSP; published as special issues of ACM SigOps *Operating Systems Review*; fall of odd-numbered years) [P4.35.06]

Operating Systems Design and Implementation (OSDI; Usenix Association, now published as special issues of ACM *SigOps Review*; fall of even-numbered years, except spring 1999 instead of fall 1998) [P4.35.U71]

Architectural Support for Programming Languages and Operating Systems (ASPLOS; published as special issues of ACM SigOps *Operating Systems Review*, SigArch *Computer Architecture News*, or *SigPlan Notices*; fall of even-numbered years) [P6.29.A7]

Applications, Technologies, Architecture, and Protocols for Computer Communication, (SigComm conference; published as special issues of ACM SigComm *Computer Communication Review*; annual) [P6.24.D31]

Principles of Distributed Computing (PODC; ACM; annual) [P4.32.D57]

Very Large Data Bases (VLDB; Morgan Kaufmann; annual) [P4.33.V4]

International Symposium on Computer Architecture (ISCA; published as special issues of ACM SigArch *Computer Architecture News*; annual) [P6.20.C6]

Less up to date, but more selective, are the journals. Often papers in these journals are revised versions of papers from the conferences listed above.

ACM Transactions on Computer Systems

ACM Transactions on Database Systems

ACM Transactions on Programming Languages and Systems

There are often good survey articles in the less technical IEEE journals:

IEEE *Computer*, *Networks*, *Communication*, *Software*

The Internet Requests for Comments (RFC's) can be reached from

`http://www.cis.ohio-state.edu/hypertext/information/rfc.html`

Rudiments of logic

Propositional logic

The basic type is `Bool`, which contains two elements `true` and `false`. Expressions in these operators (and the other ones introduced later) are called 'propositions'.

Basic operators. These are ∧ (and), ∨ (or), and ~ (not).[1] The meaning of these operators can be conveniently given by a 'truth table' which lists the value of `a op b` for each possible combination of values of `a` and `b` (the operators on the right are discussed later) along with some popular names for certain expressions and their operands.

		negation	conjunction	disjunction	equality		implication
		not	and	or			implies
a	b	~a	a ∧ b	a ∨ b	a = b	a ≠ b	a ⇒ b
T	T	F	T	T	T	F	T
T	F		F	T	F	T	F
F	T	T	F	T	F	T	T
F	F		F	F	T	F	T
name of a			conjunct	disjunct			antecedent
name of b			conjunct	disjunct			consequent

Note: In Spec we write ==> instead of the ⇒ that mathematicians use for implication. Logicians write ⊃ for implication, which looks different but is shaped like the > part of ⇒.

Since the table has only four rows, there are only 16 Boolean operators, one for each possible arrangement of `T` and `F` in a column. Most of the ones not listed don't have common names, though 'not and' is called 'nand' and 'not or' is called 'nor' by logic designers.

The ∧ and ∨ operators are
commutative and
associative and
distribute over each other.
That is, they are just like * (times) and + (plus) on integers, except that + doesn't distribute over *:

```
a + (b * c) ≠ (a + b) * (a + c)
```

but ∨ does distribute over ∧:

```
a ∨ (b ∧ c) = (a ∨ b) ∧ (a ∨ c)
```

An operator that distributes over ∧ is called 'conjunctive'; one that distributes over ∨ is called 'disjunctive'. Both ∧ and ∨ are both conjunctive and disjunctive. This takes some getting used to.

The relation between these operators and ~ is given by DeMorgan's laws (sometimes called the "bubble rule" by logic designers), which say that you can push ~ inside ∧ or ∨ (or pull it out) by flipping from one to the other:

```
~ (a ∧ b)  =  ~a ∨ ~b
~ (a ∨ b)  =  ~a ∧ ~b
```

[1] It's possible to write all three in terms of the single operator 'nor' or 'nand', but our goal is clarity, not minimality.

To put a complex expression into " disjunctive normal form" replace terms in = and ⇒ with their equivalents in ∧, ∨, and ~ (given below), use DeMorgan's laws to push all the ~'s in past ∧ and ∨ so that they apply to variables, and then distribute ∧ over ∨ so that the result looks like

$(a_1 \wedge \sim a_2 \wedge \ldots) \vee (\sim b_1 \wedge b_2 \wedge \ldots) \vee \ldots$

The disjunctive normal form is unique (up to ordering, since ∧ and ∨ are commutative). Of course, you can also distribute ∨ over ∧ to get a unique "conjunctive normal form".

If you want to find out whether two expressions are equal, one way is to put them both into disjunctive (or conjunctive) normal form, sort the terms, and see whether they are identical. Another way is to list all the possible values of the variables (if there are *n* variables, there are 2^n of them) and tabulate the values of the expressions for each of them; we saw this 'truth table' for some two-variable expressions above.

Because `Bool` is the result type of relations like `=`, you can write expressions that mix up relations with other operators in ways that are impossible for any other type. Notably

`(a = b) = ((a ∧ b) ∨ (~a ∧ ~b))`

Some people feel that the outer = in this expression is somehow different from the inner one, and write it ≡. Experience suggests, however, that this is often a harmful distinction to make.

Implication. We can define an ordering on `Bool` with `false > true`, that is, `false` is greater than `true`. The non-strict version of this ordering is called 'implication' and written ⇒ (rather than ≥ or >= as we do with other types; logicians write it ⊃, which also looks like an ordering symbol). So `(true ⇒ false) = false` (read this as: "`true` is greater than or equal to `false`" is false) but all other combinations are `true`. The expression `a ⇒ b` is pronounced "`a` implies `b`", or "if `a` then `b`".[2]

There are lots of rules for manipulating expressions containing ⇒; the most useful ones are given below. If you remember that ⇒ is an ordering you'll find it easy to remember most of the rules, but if you forget the rules or get confused, you can turn the ⇒ into ∨ by the rule

`(a ⇒ b) = ~a ∨ b`

and then just use the simpler rules for ∧, ∨, and ~. So remember this even if you forget everything else.

The point of implication is that it tells you when one proposition is stronger than another, in the sense that if the first one is true, the second is also true (because if both `a` and `a ⇒ b` are `true`, then `b` must be `true` since it can't be `false`).[3] So we use implication all the time when reasoning from premises to conclusions. Two more ways to pronounce `a ⇒ b` are "`a` is stronger than `b`" and "`b` follows from `a`". The second pronunciation suggests that it's sometimes useful to write the operands in the other order, as `b ⇐ a`, which can also be pronounced "`b` is weaker than `a`" or "`b` only if `a`"; this should be no surprise, since we do it with other orderings.

[2] It sometimes seems odd that `false` implies `b` regardless of what `b` is, but the "if ... then" form makes it clearer what is going on: if `false` is `true` you can conclude anything, but of course it isn't. A proposition that implies `false` is called 'inconsistent' because it implies anything. Obviously it's bad to think that an inconsistent proposition is true. The most likely way to get into this hole is to think that each of a collection of innocent looking propositions is true when their conjunction turns out to be inconsistent.

[3] It may also seem odd that `false > true` rather than the other way around, since `true` seems better and so should be bigger. But in fact if we want to conclude lots of things, being close to `false` is better because if `false` is true we can conclude anything, but knowing that `true` is true doesn't help at all. Strong propositions are as close to `false` as possible; this is logical brinkmanship. For example, `a ∧ b` is closer to `false` than `a` (there are more values of the variables `a` and `b` that make it `false`), and clearly we can conclude more things from it than from `a` alone.

Of course, implication has the properties we expect of an ordering:

Transitive: If `a ⇒ b` and `b ⇒ c` then `a ⇒ c`.[4]

Reflexive: `a ⇒ a`.

Anti-symmetric: If `a ⇒ b` and `b ⇒ a` then `a = b`.[5]

Furthermore, ~ reverses the sense of implication (this is called the 'contrapositive'):

`(a ⇒ b) = (~b ⇒ ~a)`

More generally, you can move a disjunct on the right to a conjunct on the left by negating it, or vice versa. Thus

`(a ⇒ b ∨ c) = (a ∧ ~b ⇒ c)`

As special cases in addition to the contrapositive we have

`(a ⇒ b) = (a ∧ ~b ⇒ false ) = ~ (a ∧ ~b) ∨ false = ~a ∨ b`

`(a ⇒ b) = (true ⇒ ~a ∨ b) = false ∨ ~a ∨ b = ~a ∨ b`

since `false` and `true` are the identities for ∨ and ∧.

We say that an operator `op` is 'monotonic' in an operand if replacing that operand with a stronger (or weaker) one makes the result stronger (or weaker). Precisely, "`op` is monotonic in its first operand" means that if `a ⇒ b` then `(a op c) ⇒ (b op c)`. Both ∧ and ∨ are monotonic; in fact, any operator that is conjunctive (distributes over ∧) is monotonic, because if `a ⇒ b` then `a = (a ∧ b)`, so

`a op c = (a ∧ b) op c = a op c ∧ b op c ⇒ b op c`

If you know what a lattice is, you will find it useful to know that the set of propositions forms a lattice with ⇒ as its ordering and (remember, think of ⇒ as "greater than or equal"):

top	= `false`		
bottom	= `true`		
meet	= ∧	least upper bound,	so `(a ∧ b) ⇒ a` and `(a ∧ b) ⇒ b`
join	= ∨	greatest lower bound,	so `a ⇒ (a ∨ b)` and `b ⇒ (a ∨ b)`

This suggests two more expressions that are equivalent to `a ⇒ b`:

`(a ⇒ b) = (a = (a ∧ b))`	'and'ing a weaker term makes no difference, because `a ⇒ b` iff `a =` least upper bound(`a`, `b`).
`(a ⇒ b) = (b = (a ∨ b))`	'or'ing a stronger term makes no difference, because `a ⇒ b` iff `b =` greatest lower bound(`a`, `b`).

Predicate logic

Propositions that have free variables, like `x < 3` or `x < 3 ⇒ x < 5`, demand a little more machinery. You can turn such a proposition into one without a free variable by substituting some value for the variable. Thus if `P(x)` is `x < 3` then `P(5)` is `5 < 3 = false`. To get rid of the free variable without substituting a value for it, you can take the 'and' or 'or' of the proposition for all the possible values of the free variable. These have special names and notation[6]:

$\forall$ `x | P(x) = P(x1) ∧ P(x2) ∧ ...`	for all `x`, `P(x)`. In Spec, `(ALL x | P(x))` or ∧ `: {x | P(x)}`

[4] We can also write this `((a ⇒ b) ∧ (b ⇒ c)) ⇒ (a ⇒ c)`.

[5] Thus `(a = b) = (a ⇒ b ∧ b ⇒ a)`, which is why `a = b` is sometimes pronounced "`a` if and only if `b`" and written "`a` iff `b`".

[6] There is no agreement on what symbol should separate the ∀ `x` or ∃ `x` from the `P(x)`. We use '|' here as Spec does, but other people use '.' or ':' or just a space, or write `(∀ x)` and `(∃ x)`. Logicians traditionally write `(x)` and `(∃ x)`.

`∃ x | P(x) = P(x₁) ∨ P(x₂) ∨ ...` there exists an `x` such that `P(x)`. In Spec, `(EXISTS x | P(x))` or `∨ : {x | P(x)}`

Here the x_i range over all the possible values of the free variables.[7] The first is called 'universal quantification'; as you can see, it corresponds to conjunction. The second is called 'existential quantification' and corresponds to disjunction. If you remember this you can easily figure out what the quantifiers do with respect to the other operators.

In particular, DeMorgan's laws generalize to quantifiers:

```
~ (∀ x | P(x)) = (∃ x | ~P(x))
~ (∃ x | P(x)) = (∀ x | ~P(x))
```

Also, because ∧ and ∨ are conjunctive and therefore monotonic, ∀ and ∃ are conjunctive and therefore monotonic.

It is not true that you can reverse the order of ∀ and ∃, but it's sometimes useful to know that having ∃ first is stronger:

```
∃ y | ∀ x | P(x, y) ⇒ ∀ x | ∃ y | P(x, y)
```

Intuitively this is clear: a `y` that works for every `x` can surely do the job for each particular `x`.

If we think of `P` as a relation, the consequent in this formula says that `P` is total (relates every `x` to some `y`). It doesn't tell us anything about how to find a `y` that is related to `x`. As computer scientists, we like to be able to compute things, so we prefer to have a function that computes `y`, or the set of `y`'s, from `x`. This is called a 'Skolem function'; in Spec you write `P.func` (or `P.setF` for the set). `P.func` is total if `P` is total. Or, to turn this around, if we have a total function `f` such that `∀ x | P(x, f(x))`, then certainly `∀ x | ∃ y | P(x, y)`; in fact, `y = f(x)` will do. Amazing.

Summary of logic

The ∧ and ∨ operators are commutative and associative and distribute over each other.

DeMorgan's laws: `~ (a ∧ b) = ~a ∨ ~b`
`~ (a ∨ b) = ~a ∧ ~b`

Any expression has a unique (up to ordering) disjunctive normal form in which ∨ combines terms in which ∨ combines (possibly negated) variables: `(a₁ ∧ ~a₂ ∧ ...) ∨ (~b₁ ∧ b₂ ∧ ...) ∨ ...`

Implication: `(a ⇒ b) = ~a ∨ b`

Implication is the ordering in a lattice (a partially ordered set in which every subset has a least upper and a greatest lower bound) with

top	= `false`	so `false ⇒ true`
bottom	= `true`	
meet	= ∧	least upper bound, so `(a ∧ b) ⇒ a`
join	= ∨	greatest lower bound, so `a ⇒ (a ∨ b)`

For all `x`, `P(x)`:

```
∀ x | P(x) = P(x₁) ∧ P(x₂) ∧ ...
```

There exists an `x` such that `P(x)`:

```
∃ x | P(x) = P(x₁) ∨ P(x₂) ∨ ...
```

[7] In general this might not be a countable set, so the conjunction and disjunction are written in a somewhat misleading way, but this complication won't make any difference to us.

Index for logic

~, 6
==>, 6
ALL, 9
and, 6
antecedent, 6
Anti-symmetric, 8
associative, 6
bottom, 8
commutative, 6
conjunction, 6
conjunctive, 6
consequent, 6
contrapositive, 8
DeMorgan's laws, 7, 9
disjunction, 6
disjunctive, 6
distribute, 6
existential quantification, 9
EXISTS, 9
follows from, 7
free variables, 8
greatest lower bound, 8
if a then b, 7
implication, 6, 7
join, 8
lattice, 8
least upper bound, 8
meet, 8
monotonic, 8
negation, 6
not, 6
only if, 7
operators, 6
or, 6
ordering on Bool, 7
predicate logic, 8
propositions, 6
quantifiers, 9
reflexive, 8
Skolem function, 9
stronger than, 7
top, 8
transitive, 8
truth table, 6
universal quantification, 9
weaker than, 7