

1. Course Information

Staff

Faculty

Butler Lampson

Martin Rinard

Office Hours

Messrs. Lampson and Rinard will arrange individual appointments. The TA will hold scheduled office hours at a time to be announced. In addition to holding regularly scheduled office hours, the TA will also be available by appointment.

Lectures and handouts

Lectures are held on Tuesdays and Thursdays from 1:00 to 2:30PM.

Messrs. Lampson and Rinard 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.

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 in class the following Wednesday. 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 5, 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 17, 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 Schedule

Date	No	By	HO	Topic	PS out	PS due
Tues., Feb. 5	1	L		Overview. The Spec language. State machine semantics. Examples of specifications and code. 1 Course information 2 Background 3 Introduction to Spec 4 Spec reference manual 5 Examples of specs and code	1	
Thurs., Feb. 7	2	L		Spec and code for sequential programs. Correctness notions and proofs. Proof methods: abstraction functions and invariants. 6 Abstraction functions		
Tues., Feb. 12	3	L		File systems 1: Disks, simple sequential file system, caching, logs for crash recovery. 7 Disks and file systems	2	1
Thurs., Feb. 14	4	L		File systems 2: Copying file system.		
Tues., Feb. 19				No class, Monday schedule from Presidents' Day		
Thurs., Feb. 21	5	R		Proof methods: History and prophecy variables; abstraction relations. 8 History variables	3	2
Tues., Feb. 26	6	R		Semantics and proofs: Formal sequential semantics of Spec. 9 Atomic semantics of Spec		
Thurs., Feb. 28	7	L		Performance: How to get it, how to analyze it. 10 Performance 11 Paper: Michael Schroeder and Michael Burrows, Performance of Firefly RPC, <i>ACM Transactions on Computer Systems</i> 8 , 1, February 1990, pp 1-17.	4	3
Tues., Mar. 5	8	L		Naming: Specs, variations, and examples of hierarchical naming. 12 Naming 13 Paper: David Gifford et al, Semantic file systems, <i>Proc. 13th ACM Symposium on Operating System Principles</i> , October 1991, pp 16-25.		Form groups
Thurs., Mar. 7	9	R		Concurrency 1: Practical concurrency, easy and hard. Easy concurrency using locks and condition variables. Problems with it: scheduling, deadlock.	5	4

Date	No	By	HO	Topic	PS out	PS due
				14 Practical concurrency		
				15 Concurrent disks		
				16 Paper: Andrew Birrell, An Introduction to Programming with Threads, Digital Systems Research Center Report 35, January 1989.		
Tues., Mar. 12	10	R		Concurrency 2: Concurrency in Spec: threads and non-atomic semantics. Big atomic actions. Safety and liveness. Examples of concurrency.		
				17 Formal concurrency		
Thurs., Mar. 14	11	R		Concurrency 3: Proving correctness of concurrent programs: assertional proofs, model checking		5
Tues., Mar. 19	12	R		Distributed consensus 1. Paxos algorithm for asynchronous consensus in the presence of faults.		
Thurs., Mar. 21	13	L		Distributed consensus 2.		Early proposals
				18 Consensus		
Mar. 25-29				Spring Break		
Tues., Apr. 2	14	R		Sequential transactions with caching.		
				19 Sequential transactions		
Thurs., Apr. 4	15	R		Concurrent transactions: Specs for serializability. Ways to code the specs.		Late proposals
				20 Concurrent transactions		
Tues., Apr. 9	16	R		Distributed transactions: Commit as a consensus problem. Two-phase commit. Optimizations.		Late interim reports
				27 Distributed transactions		
Thurs., Apr. 11	17	R		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, <i>IEEE Journal on Selected Areas in Communications</i> 9, 8, October 1991, pp 1318-1335.		
				23 Networks: Links and switches		
Tues., Apr. 16				Patriot's Day, no class.		

Date	No	By	HO	Topic	PS out	PS due
Thurs., Apr. 18	18	L		Networks 2: Links cont'd: Ethernet. Token Rings. Switches. Coding switches. Routing. Learning topologies and establishing routes.		
Tues., Apr. 23	19	L		Networks 3: Network objects and remote procedure call (RPC).		Early interim reports
				24 Network objects		
				25 Paper: Andrew Birrell et al., Network objects, <i>Proc. 14th ACM Symposium on Operating Systems Principles</i> , Asheville, NC, December 1993.		
Thurs., Apr. 25	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 <i>Distributed Systems</i> , ed. S. Mullender, Addison-Wesley, 1993, pp 251-281.		
Tues., Apr. 30	21	L		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 <i>Transaction Processing: Concepts and Techniques</i> , Morgan Kaufmann, 1993, pp 93-156.		
Thurs., May 2	22	R		Caching: Maintaining coherent memory. Broadcast (snoopy) and directory protocols. Examples: multiprocessors, distributed shared memory, distributed file systems.		
				30 Concurrent caching		
Tues., May 7	23			Early project presentations		
Thurs., May 9	24			Early project presentations		
Tues., May 14	25			Late project presentations		
Thurs., May 16	26			Late project presentations		
Fri., May 17				Final reports due		
May 20-24				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.

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 LCS reading room. The cryptic strings in brackets are call numbers there. You can also find the last few years of 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 \wedge (and), \vee (or), and \sim (not).¹ The meaning of these operators can be conveniently given by a ‘truth table’ which lists the value of $a \text{ op } b$ 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	
a	b	not	and	or	$a = b$	$a \neq b$	implies
		$\sim a$	$a \wedge b$	$a \vee b$			$a \Rightarrow b$
T	T	F	T	T	T	F	T
T	F	T	F	T	F	T	F
F	T	T	F	T	F	T	T
F	F	T	F	F	T	F	T
name of a			conjunct	disjunct			antecedent
name of b			conjunct	disjunct			consequent

Note: In Spec we write \Rightarrow instead of the \Rightarrow that mathematicians use for implication. Logicians write \supset for implication, which looks different but is shaped like the $>$ part of \Rightarrow .

In case you have an expression that you can’t simplify, you can always work out its truth value by exhaustively enumerating the cases in truth table style. 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 \wedge and \vee 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 $*$:

$$a + (b * c) \neq (a + b) * (a + c)$$

but \vee does distribute over \wedge :

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

An operator that distributes over \wedge is called ‘conjunctive’; one that distributes over \vee is called ‘disjunctive’. So both \wedge and \vee are both conjunctive and disjunctive. This takes some getting used to.

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

The relation between these operators and \sim is given by DeMorgan's laws (sometimes called the "bubble rule" by logic designers), which say that you can push \sim inside \wedge or \vee by flipping from one to the other:

$$\begin{aligned}\sim (a \wedge b) &= \sim a \vee \sim b \\ \sim (a \vee b) &= \sim a \wedge \sim b\end{aligned}$$

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

$$(a = b) = ((a \wedge b) \vee (\sim a \wedge \sim b))$$

Some people feel that the outer $=$ in this expression is somehow different from the inner one, and write it \equiv . 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 \Rightarrow (rather than \geq or $>=$ as we do with other types; logicians write it \supset , which also looks like an ordering symbol). So $(\text{true} \Rightarrow \text{false}) = \text{false}$ (read this as: "true is greater than or equal to false" is false) but all other combinations are `true`. The expression $a \Rightarrow b$ is pronounced "a implies b", or "if a then b".²

There are lots of rules for manipulating expressions containing \Rightarrow ; the most useful ones are given below. If you remember that \Rightarrow 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 \Rightarrow into \vee by the rule

$$(a \Rightarrow b) = \sim a \vee b$$

and then just use the simpler rules for \wedge , \vee , and \sim . 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 $a \Rightarrow b$ 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 $a \Rightarrow b$ 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 $b \Leftarrow a$, 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.

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

Transitive: If $a \Rightarrow b$ and $b \Rightarrow c$ then $a \Rightarrow c$.⁴

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, $a \wedge b$ 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.

⁴ We can also write this $((a \Rightarrow b) \wedge (b \Rightarrow c)) \Rightarrow (a \Rightarrow c)$.

Reflexive: $a \Rightarrow a$.

Anti-symmetric: If $a \Rightarrow b$ and $b \Rightarrow a$ then $a = b$.⁵

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

$$(a \Rightarrow b) = (\sim b \Rightarrow \sim a)$$

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

Thus

$$(a \Rightarrow b \vee c) = (a \wedge \sim b \Rightarrow c)$$

As special cases in addition to the contrapositive we have

$$\begin{aligned}(a \Rightarrow b) &= (a \wedge \sim b \Rightarrow \text{false}) = \sim (a \wedge \sim b) \vee \text{false} = \sim a \vee b \\ (a \Rightarrow b) &= (\text{true} \Rightarrow \sim a \vee b) = \text{false} \vee \sim a \vee b = \sim a \vee b\end{aligned}$$

since `false` and `true` are the identities for \vee and \wedge .

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 $a \Rightarrow b$ then $(a \text{ op } c) \Rightarrow (b \text{ op } c)$. Both \wedge and \vee are monotonic; in fact, any conjunctive operator is monotonic, because if $a \Rightarrow b$ then $a = (a \wedge b)$, so $a \text{ op } c = (a \wedge b) \text{ op } c = a \text{ op } c \wedge b \text{ op } c \Rightarrow b \text{ op } c$.

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

$$\begin{array}{ll} \text{top} &= \text{false} \\ \text{bottom} &= \text{true} \\ \text{meet} &= \wedge && \text{least upper bound, so } (a \wedge b) \Rightarrow a \text{ and } (a \wedge b) \Rightarrow b \\ \text{join} &= \vee && \text{greatest lower bound, so } a \Rightarrow (a \vee b) \text{ and } b \Rightarrow (a \vee b) \end{array}$$

This suggests two more expressions that are equivalent to $a \Rightarrow b$:

$$\begin{aligned}(a \Rightarrow b) &= (a = (a \wedge b)) && \text{'and'ing a weaker term makes no difference,} \\ &&& \text{because } a \Rightarrow b \text{ iff } a = \text{least upper bound}(a, b). \\ (a \Rightarrow b) &= (b = (a \vee b)) && \text{'or'ing a stronger term makes no difference,} \\ &&& \text{because } a \Rightarrow b \text{ iff } b = \text{greatest lower bound}(a, b).\end{aligned}$$

Predicate logic

Propositions that have free variables, like $x < 3$ or $x < 3 \Rightarrow x < 5$, 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 $5 < 3 = \text{false}$. 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 notation⁶:

$$\begin{aligned}\forall x \mid P(x) &= P(x1) \wedge P(x2) \wedge \dots && \text{for all } x, P(x). \text{ In Spec,} \\ &&& (\text{ALL } x \mid P(x)) \text{ or } \wedge : \{x \mid P(x)\}\end{aligned}$$

⁵ Thus $(a = b) = (a \Rightarrow b \wedge b \Rightarrow a)$, 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 $\forall x$ or $\exists x$ from the $P(x)$. We use '|' here as Spec does, but other people use '.', or ':' or just a space, or write $(\forall x)$ and $(\exists x)$. Logicians traditionally write (x) and $(\exists x)$.

$\exists x \mid P(x) = P(x_1) \vee P(x_2) \vee \dots$ there exists an x such that $P(x)$. In Spec,
(EXISTS $x \mid P(x)$) or $\vee : \{x \mid P(x)\}$

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:

$$\begin{aligned} \sim (\forall x \mid P(x)) &= (\exists x \mid \sim P(x)) \\ \sim (\exists x \mid P(x)) &= (\forall x \mid \sim P(x)) \end{aligned}$$

Also, because \wedge and \vee are conjunctive and therefore monotonic, \forall and \exists are conjunctive and therefore monotonic.

It is not true that you can reverse the order of \forall and \exists , but it’s sometimes useful to know that having \exists first is stronger:

$$\exists y \mid \forall x \mid P(x, y) \Rightarrow \forall x \mid \exists y \mid P(x, y)$$

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 $\forall x \mid P(x, f(x))$, then certainly $\forall x \mid \exists y \mid P(x, y)$; in fact, $y = f(x)$ will do. Amazing.

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.

Summary of logic

The \wedge and \vee operators are commutative and associative and distribute over each other.

DeMorgan’s laws: $\sim (a \wedge b) = \sim a \vee \sim b$
 $\sim (a \vee b) = \sim a \wedge \sim b$

Implication: $(a \Rightarrow b) = \sim a \vee b$

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 \Rightarrow true
bottom	= true	
meet	= \wedge	least upper bound, so $(a \wedge b) \Rightarrow a$
join	= \vee	greatest lower bound, so $a \Rightarrow (a \vee b)$

For all x , $P(x)$:

$$\forall x \mid P(x) = P(x_1) \wedge P(x_2) \wedge \dots$$

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

$$\exists x \mid P(x) = P(x_1) \vee P(x_2) \vee \dots$$

Index for logic

\sim , 6	meet, 8
\Rightarrow , 6	monotonic, 8
ALL, 9	negation, 6
and, 6	not, 6
antecedent, 6	only if, 7
Anti-symmetric, 8	operators, 6
associative, 6	or, 6
bottom, 8	ordering on Bool, 7
commutative, 6	predicate logic, 8
conjunction, 6	propositions, 6
conjunctive, 6	quantifiers, 9
consequent, 6	reflexive, 8
contrapositive, 8	Skolem function, 9
DeMorgan’s laws, 7, 9	stronger than, 7
disjunction, 6	top, 8
disjunctive, 6	transitive, 8
distribute, 6	truth table, 6
existential quantification, 9	universal quantification, 9
EXISTS, 9	weaker than, 7
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	

3. Introduction to Spec

This handout explains what the Spec language is for, how to use it effectively, and how it differs from a programming language like C, Pascal, Clu, Java, or Scheme. Spec is very different from these languages, but it is also much simpler. Its meaning is clearer and Spec programs are more succinct and less burdened with trivial details. The handout also introduces the main constructs that are likely to be unfamiliar to a programmer. You will probably find it worthwhile to read it over more than once, until those constructs are familiar. Don't miss the one-page summary of spec at the end. The handout also has an index.

Spec is a language for writing precise descriptions of digital systems, both sequential and concurrent. In Spec you can write something that differs from practical code (for instance, code written in C) only in minor details of syntax. This sort of thing is usually called a program. Or you can write a very high level description of the behavior of a system, usually called a specification. A good specification is almost always quite different from a good program. You can use Spec to write either one, but not the same *style* of Spec. The flexibility of the language means that you need to know the purpose of your Spec in order to write it well.

Most people know a lot more about writing programs than about writing specs, so this introduction emphasizes how Spec differs from a programming language and how to use it to write good specs. It does not attempt to be either complete or precise, but other handouts fill these needs. The *Spec Reference Manual* (handout 4) describes the language completely; it gives the syntax of Spec precisely and the semantics informally. *Atomic Semantics of Spec* (handout 9) describes precisely the meaning of an atomic command; here 'precisely' means that you should be able to get an unambiguous answer to any question. The section "Non-Atomic Semantics of Spec" in handout 17 on formal concurrency describes the meaning of a non-atomic command.

Spec's notation for commands, that is, for changing the state, is derived from Edsger Dijkstra's guarded commands (E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976) as extended by Greg Nelson (G. Nelson, A generalization of Dijkstra's calculus, *ACM TOPLAS* **11**, 4, Oct. 1989, pp 517-561). The notation for expressions is derived from mathematics.

This handout starts with a discussion of specifications and how to write them, with many small examples of Spec. Then there is an outline of the Spec language, followed by three extended examples of specs and code. At the end are two handy tear-out one-page summaries, one of the language and one of the official POCS strategy for writing specs and code.

What is a specification for?

The purpose of a specification is to communicate precisely all the essential facts about the behavior of a system. The important words in this sentence are:

<i>communicate</i>	The spec should tell both the client and the implementer what each needs to know.
<i>precisely</i>	We should be able to prove theorems or compile machine instructions based on the spec.
<i>essential</i>	Unnecessary requirements in the spec may confuse the client or make it more expensive to implement the system.
<i>behavior</i>	We need to know exactly what we mean by the behavior of the system.

Communication

Spec mediates communication between the client of the system and its implementer. One way to view the spec is as a contract between these parties:

The client agrees to depend only on the system behavior expressed in the spec; in return it only has to read the spec, and it can count on the implementer to provide a system that actually does behave as the spec says it should.

The implementer agrees to provide a system that behaves according to the spec; in return it is free to arrange the internals of the system however it likes, and it does not have to deliver anything not laid down in the spec.

Usually the implementer of a spec is a programmer, and the client is another programmer. Usually the implementer of a program is a compiler or a computer, and the client is a programmer.

Usually the system that the implementer provides is called an implementation, but in this course we will call it *code* for short.

Behavior

What do we mean by behavior? In real life a spec defines not only the functional behavior of the system, but also its performance, cost, reliability, availability, size, weight, etc. In this course we will deal with these matters informally if at all. The Spec language doesn't help much with them.

Spec is concerned only with the possible state transitions of the system, on the theory that the possible state transitions tell the complete story of the functional behavior of a digital system. So we make the following definitions:

A *state* is the values of a set of names (for instance, `x=3`, `color=red`).

A *history* is a sequence of states such that each pair of adjacent states is a transition of the system (for instance, $x=1; x=2; x=5$ is the history if the initial state is $x=1$ and the transitions are “if $x = 1$ then $x := x + 1$ ” and “if $x = 2$ then $x := 2 * x + 1$ ”).

A *behavior* is a set of histories (a non-deterministic system can have more than one history).

How can we specify a behavior?

One way to do this is to just write down all the histories in the behavior. For example, if the state just consists of a single integer, we might write

```

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
...
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
...
1 2 3 4 5 1 2 3 1 2 3 4 5 6 7 8 9 10

```

The example reveals two problems with this approach:

The sequences are long, and there are a lot of them, so it takes a lot of space to write them down. In fact, in most cases of interest the sequences are infinite, so we can't actually write them down.

It isn't too clear from looking at such a set of sequences what is really going on.

Another description of this set of sequences from which these examples are drawn is “18 integers, each one either 1 or one more than the preceding one.” This is concise and understandable, but it is not formal enough either for mathematical reasoning or for directions to a computer.

Precise

In Spec the set of sequences can be described in many ways, for example, by the expression

```

{s: SEQ Int | s.size = 18
  /\ (ALL i: Int | 0 <= i /\ i < s.size ==>
    s(i) = 1 \/ (i > 0 /\ s(i) = s(i-1) + 1)) }

```

Here the expression in $\{ \dots \}$ is very close to the usual mathematical notation for defining a set. Read it as “The set of all s which are sequences of integers such that $s.size = 18$ and ...”. Spec sequences are indexed from 0. The $(ALL \dots)$ is a universally quantified predicate, and $==>$ stands for implication, since Spec uses the more familiar $=>$ for ‘then’ in a guarded command. Throughout Spec the ‘|’ symbol separates a declaration of some new names and their types from the scope in which they are meaningful.

Alternatively, here is a state machine that generates the sequences we want as the successive values of the variable i . We specify the transitions of the machine by starting with primitive *assignment commands* and putting them together with a few kinds of compound commands. Each command specifies a set of possible transitions.

```

VAR i, j |
  << i := 1; j := 1 >> ;
  DO << j < 18 => BEGIN i := 1 [] i := i + 1 END; j := j + 1 >> OD

```

Here there is a good deal of new notation, in addition to the familiar semicolons, assignments, and plus signs.

$VAR i, j |$ introduces the local variables i and j with arbitrary values. Because $;$ binds more tightly than $|$, the scope of the variables is the rest of the example.

The $\langle \dots \rangle$ brackets delimit the atomic actions or transitions of the state machine. All the changes inside these brackets happen as one transition of the state machine.

$j < 18 => \dots$ is a transition that can only happen when $j < 18$. Read it as “if $j < 18$ then ...”. The $j < 18$ is called a *guard*. If the guard is false, we say that the entire command *fails*.

$i := 1 [] i := i + 1$ is a *non-deterministic* transition which can either set i to 1 or increment it. Read $[]$ as ‘or’.

The $BEGIN \dots END$ brackets are just brackets for commands, like $\{ \dots \}$ in C. They are there because $=>$ binds more tightly than the $[]$ operator inside the brackets; without them the meaning would be “either set i to 1 if $j < 18$ or increment i and j unconditionally”.

Finally, the $DO \dots OD$ brackets mean: repeat the \dots transition as long as possible. Eventually j becomes 18 and the guard becomes false, so the command inside the $DO \dots OD$ fails and can no longer happen.

The expression approach is better when it works naturally, as this example suggests, so Spec has lots of facilities for describing values: sequences, sets, and functions as well as integers and booleans. Usually, however, the sequences we want are too complicated to be conveniently described by an expression; a state machine can describe them much more easily.

State machines can be written in many different ways. When each transition involves only simple expressions and changes only a single integer or boolean state variable, we think of the state machine as a program, since we can easily make a computer exhibit this behavior. When there are transitions that change many variables, non-deterministic transitions, big values like sequences or functions, or expressions with quantifiers, we think of the state machine as a spec, since it may be much easier to understand and reason about it, but difficult to make a computer exhibit this behavior. In other words, large atomic actions, non-determinism, and expressions that compute sequences or functions are hard to code. It may take a good deal of ingenuity to find code that has the same behavior but uses only the small, deterministic atomic actions and simple expressions that are easy for the computer.

Essential

The hardest thing for most people to learn about writing specs is that *a spec is not a program*. A spec defines the behavior of a system, but unlike a program it need not, and usually should not, give any practical method for producing this behavior. Furthermore, it should pin down the

behavior of the system only enough to meet the client’s needs. Details in the spec that the client doesn’t need can only make trouble for the implementer.

The example we just saw is too artificial to illustrate this point. To learn more about the difference between a spec and code consider the following:

```
CONST eps := 10**-8
APROC SquareRoot0(x: Real) -> Real =
  << VAR y : Real | Abs(x - y*y) < eps => RET y >>
```

(Spec as described in the reference manual doesn’t have a `Real` data type, but we’ll add it for the purpose of this example.)

The combination of `VAR` and `=>` is a very common Spec idiom; read it as “choose a `y` such that `Abs(x - y*y) < eps` and do `RET y`”. Why is this the meaning? The `VAR` makes a choice of any `Real` as the value of `y`, but the entire transition on the second line cannot occur unless the guard is true. The result is that the choice is restricted to a value that satisfies the guard.

What can we learn from this example? First, the result of `SquareRoot0(x)` is not completely determined by the value of `x`; any result whose square is within `eps` of `x` is possible. This is why `SquareRoot0` is written as a procedure rather than a function; the result of a function has to be determined by the arguments and the current state, so that the value of an expression like `f(x) = f(x)` will be true. In other words, `SquareRoot0` is *non-deterministic*.

Why did we write it that way? First of all, there might not be any `Real` (that is, any floating-point number of the kind used to represent `Real`) whose square exactly equals `x`.¹ Second, we may not want to pay for code that gives the closest possible answer. Instead, we may settle for a less accurate answer in the hope of getting the answer faster.

You have to make sure you know what you are doing, though. This spec allows a negative result, which is perhaps not what we really wanted. We could have written (highlighting changes with boxes):

```
APROC SquareRoot1(x: Real) -> Real =
  << VAR y : Real | y >= 0 /\ Abs(x - y*y) < eps => RET y >>
```

to rule that out. Also, the spec produces no result if `x < 0`, which means that `SquareRoot1(-1)` will fail (see the section on commands for a discussion of failure). We might prefer a total function that raises an exception:

```
APROC SquareRoot2(x: Real) -> Real RAISES {undefined} =
  << x >= 0 => VAR y : Real | y >= 0 /\ Abs(x - y*y) < eps => RET y
[*] RAISE undefined >>
```

The `[*]` is ‘else’; it does its second operand iff the first one fails. Exceptions in Spec are much like exceptions in CLU. An exception is contagious: once started by a `RAISE` it causes any

containing expression or command to yield the same exception, until it runs into an exception handler (not shown here). The `RAISES` clause of a routine declaration must list all the exceptions that the procedure body can generate, either by `RAISES` or by invoking another routine.

Code for this spec would look quite different from the spec itself. Instead of the existential quantifier implied by the `VAR y`, it would have an algorithm for finding `y`, for instance, Newton’s method. In the algorithm you would only see operations that have obvious codes in terms of the load, store, arithmetic, and test instructions of a computer. Probably the code would be deterministic.

Another way to write these specs is as functions that return the set of possible answers. Thus

```
FUNC SquareRoots1(x: Real) -> SET Real =
  RET {y : Real | y >= 0 /\ Abs(x - y*y) < eps}
```

Note that the form inside the `{...}` set constructor is the same as the guard on the `RET`. To get a single result you can use the set’s `choose` method: `SquareRoots1(2).choose`.²

In the next section we give an outline of the Spec language. Following that are three extended examples of specs and code for fairly realistic systems. At the end is a one-page summary of the language.

An outline of the Spec language

The Spec language has two main parts:

- An *expression* describes how to compute a result (a value or an exception) as a function of other values: either literal constants or the current values of state variables.
- A *command* describes possible transitions of the state variables. Another way of saying this is that a command is a relation on states: it allows a transition from `s1` to `s2` iff it relates `s1` to `s2`.

Both are based on the *state*, which in Spec is a mapping from names to values. The names are called state variables or simply variables: in the sequence example above they are `i` and `j`. Actually a command relates states to *outcomes*; an outcome is either a state (a normal outcome) or a state together with an exception (an exceptional outcome).

There are two kinds of commands:

- An *atomic* command describes a set of possible transitions, or equivalently, a set of pairs of states, or a relation between states. For instance, the command `<< i := i + 1 >>` describes the transitions `i=1→i=2`, `i=2→i=3`, etc. (Actually, many transitions are summarized by `i=1→i=2`, for instance, `(i=1, j=1)→(i=2, j=1)` and `(i=1, j=15)→(i=2, j=15)`). If a command allows more than one transition from a given state we say it is non-deterministic.

² `r := SquareRoots1(x).choose` (using the function) is almost the same as `r := SquareRoot1(x)` (using the procedure). The difference is that because `choose` is a function it always returns the same element (even though we don’t know in advance which one) when given the same set, and hence when `SquareRoots1` is given the same argument. The procedure, on the other hand, is non-deterministic and can return different values on successive calls.

For instance, on page 3 the command `BEGIN i := 1 [] i := i + 1 END` allows the transitions $i=2 \rightarrow i=1$ and $i=2 \rightarrow i=3$.

- A *non-atomic* command describes a set of *sequences* of states (by contrast with the set of pairs for an atomic command). More on this below.

A sequential program, in which we are only interested in the initial and final states, can be described by an atomic command.

The meaning of an expression, which is a function from states to values (or exceptions), is much simpler than the meaning of an atomic command, which is a relation between states, for two reasons:

- The expression yields a single value rather than an entire state.
- The expression yields at most one value, whereas a non-deterministic command can yield many final states.

A atomic command is still simple, much simpler than a non-atomic command, because:

- Taken in isolation, the meaning of a non-atomic command is a relation between an initial state and a history. Again, many histories can stem from a single initial state.
- The meaning of the composition of two non-atomic commands is not any simple combination of their relations, such as the union, because the commands can interact if they share any variables that change.

These considerations lead us to describe the meaning of a non-atomic command by breaking it down into its atomic subcommands and connecting these up with a new state variable called a program counter. The details are somewhat complicated; they are sketched in the discussion of atomicity below, and described in *handout 17* on formal concurrency.

The moral of all this is that you should use the simpler parts of the language as much as possible: expressions rather than atomic commands, and atomic commands rather than non-atomic ones. To encourage this style, Spec has a lot of syntax and built-in types and functions that make it easy to write expressions clearly and concisely. You can write many things in a single Spec expression that would require a number of C statements, or even a loop. Of course, code with a lot of concurrency will necessarily have more non-atomic commands, but this complication should be put off as long as possible.

Organizing the program

In addition to the expressions and commands that are the core of the language, Spec has four other mechanisms that are useful for organizing your program and making it easier to understand.

- A *routine* is a named computation with parameters, in other words, an abstraction of the computation. Parameters are passed by value. There are four kinds of routine:

A *function* (defined with `FUNC`) is an abstraction of an expression.

An *atomic procedure* (defined with `APROC`) is an abstraction of an atomic command.

A *general procedure* (defined with `PROC`) is an abstraction of a non-atomic command.

A *thread* (defined with `THREAD`) is the way to introduce concurrency.

- A *type* is a highly stylized assertion about the set of values that a name or expression can assume. A type is also a convenient way to group and name a collection of routines, called its *methods*, that operate on values in that set.
- An *exception* is a way to report an unusual outcome.
- A *module* is a way to structure the name space into a two-level hierarchy. An identifier `i` declared in a module `m` has the name `m.i` throughout the program. A *class* is a module that can be instantiated many times to create many objects.

A Spec program is some global declarations of variables, routines, types, and exceptions, plus a set of modules each of which declares some variables, routines, types, and exceptions.

The next two sections describe things about Spec's expressions and commands that may be new to you. It doesn't answer every question about Spec; for those answers, read the reference manual and the handouts on Spec semantics. There is a one-page summary at the end of this handout.

Expressions, types, and functions

Expressions are for computing functions of the state. A Spec expression is a constant, a variable, or an invocation of a function on an argument that is some sub-expression. The values of these expressions are the constant, the current value of the variable, or the value of the function at the value of the argument. There are no side-effects; those are the province of commands. There is quite a bit of syntactic sugar for function invocations. An expression may be undefined in a state; if a simple command evaluates an undefined expression, the command fails (see below).

A Spec type defines two things:

A set of values; we say that a value *has* the type if it's in the set. The sets are not disjoint.

A set of functions called the *methods* of the type. There is convenient syntax `v.m` for invoking method `m` on a value `v` of the type. A method `m` of type `T` is lifted to functions $U \rightarrow T$, sets of `T`'s, and relations from `U` to `T` in the obvious way, unless overridden by a different `m` in the definition of the higher type.

Spec is strongly typed. This means that you are supposed to declare the types of your variables, just as you do in Pascal or CLU. In return the language defines a type for every expression³ and ensures that the value of the expression always has that type. In particular, the value of a variable always has the declared type. You should think of a type declaration as a stylized comment that has a precise meaning and could be checked mechanically.

³ Note that a value may have many types, but a variable or an expression has exactly one type: for a variable, it's the declared type, and for a complex expression it's the result type of the top-level function in the expression.

If `FOO` is a type, you can omit it in a declaration of the identifiers `foo`, `foo1`, `foo'` etc. Thus

```
VAR int1, bool2, char'' | ...
```

is short for

```
VAR int1: Int, bool2: Bool, char'': Char | ...
```

`Spec` has the usual types: `Int`, `Nat` (non-negative `Int`), `Bool`, functions, sets, records, tuples, and variable-length arrays called sequences. A sequence is a function whose domain is $\{0, 1, \dots, n-1\}$ for some n . In addition to the usual functions like `+` and `\`, `Spec` also has some less usual operations on these types, which are valuable when you want to suppress code detail: constructors and combinations.

You can make a type with fewer values using `SUCHTHAT`. For example,

```
TYPE T = Int SUCHTHAT (\ i: Int | 0 <= i /\ i <= 4)
```

has the value set $\{0, 1, 2, 3, 4\}$. Here the $(\ \dots)$ is a lambda expression (with $\$ for λ) that defines a function from `Int` to `Bool`, and a value has type `T` if it's an `Int` and the function maps it to `true`.

Section 5 of the reference manual describes expressions and lists all the built-in operators. You should read the list, which also gives their precedence and has pointers to explanations of their meaning. Section 4 describes the types. Section 9 defines the built-in methods for sequences, sets, and functions; you should read it over so that you know the vocabulary.

Constructors

Constructors for functions, sets, and sequences make it easy to toss large values around. For instance, you can describe a database as a function `db` from names to data records with two fields:

```
TYPE DB = (String -> Entry)
TYPE Entry = [salary: Int, birthdate: Int]
VAR db := DB{}
```

Here `db` is initialized using a function constructor whose value is a function undefined everywhere. The type can be omitted in a variable declaration when the variable is initialized; it is taken to be the type of the initializing expression. The type can also be omitted when it is the upper case version of the variable name, `DB` in this example.

Now you can make an entry with

```
db := db{ "Smith" -> Entry{salary := 23000, birthdate := 1955} }
```

using another function constructor. The value of the constructor is a function that is the same as `db` except at the argument `"Smith"`, where it has the value `Entry{...}`, which is a record constructor. The assignment could also be written

```
db("Smith") := Entry{salary := 23000, birthdate := 1955}
```

which changes the value of the `db` function at `"Smith"` without changing it anywhere else. This is actually a shorthand for the previous assignment. You can omit the field names if you like, so that

```
db("Smith") := Entry{23000, 1955}
```

has the same meaning as the previous assignment. Obviously this shorthand is less readable and more error-prone, so use it with discretion. Another way to write this assignment is

```
db("Smith").salary := 23000; db("Smith").birthdate := 1955
```

The set of names in the database can be expressed by a set constructor. It is just

```
{n: String | db!n},
```

in other words, the set of all the strings for which the `db` function is defined (`!` is the ‘is-defined’ operator; that is, $f!x$ is true iff f is defined at x). Read this “the set of strings n such that `db!n`”. You can also write it as `db.dom`, the domain of `db`; section 9 of the reference manual defines lots of useful built in methods for functions, sets, and sequences. It’s important to realize that you can freely use large (possibly infinite) values such as the `db` function. You are writing a spec, and you don’t need to worry about whether the compiler is clever enough to turn an expensive-looking manipulation of a large object into a cheap incremental update. That’s the implementer’s problem (so you may have to worry about whether she is clever enough).

If we wanted the set of lengths of the names, we would write

```
{n: String | db!n | n.size}
```

This three part set constructor contains i if and only if there exists an n such that `db!n` and $i = n.size$. So $\{n: String | db!n\}$ is short for $\{n: String | db!n | n\}$. You can introduce more than one name, in which case the third part defaults to the last name. For example, if we represent a directed graph by a function on pairs of nodes that returns `true` when there’s an edge from the first to the second, then

```
{n1: Node, n2: Node | graph(n1, n2) | n2}
```

is the set of nodes that are the target of an edge, and the `| n2` could be omitted. This is just the range `graph.rng` of the relation `graph` on nodes.

Following standard mathematical notation, you can also write

```
{f: IN openFiles | f.modified}
```

to get the set of all open, modified files. This is equivalent to

```
{f: File | f IN openFiles /\ f.modified}
```

because if s is a SET T , then `IN s` is a type whose values are the T ’s in s ; in fact, it’s the type `T SUCHTHAT (\ t | t IN s)`. This form also works for sequences, where the second operand of `IN` provides the ordering. So if s is a sequence of integers, $\{x: IN s | x > 0\}$ is the positive ones, $\{x: IN s | x > 0 | x * x\}$ is the squares of the positive ones, and $\{x: IN s | | x * x\}$ is the squares of all the integers, because an omitted predicate defaults to `true`.⁴

To get sequences that are more complicated you can use sequence generators with `BY` and `WHILE`.

```
{i := 1 BY i + 1 WHILE i <= 5 | true | i}
```

is $\{1, 2, 3, 4, 5\}$; the second and third parts could be omitted. This is just like the “for” construction in C. An omitted `WHILE` defaults to `true`, and an omitted `:=` defaults to an arbitrary choice for the initial value. If you write several generators, each variable gets a new value for each value produced, but the second and later variables are initialized first. So to get the sums of successive pairs of elements of s , write

```
{x := s BY x.tail WHILE x.size > 1 | | x(0) + x(1)}
```

To get the sequence of partial sums of s , write (eliding `| | sum` at the end)

```
{x: IN s, sum := 0 BY sum + x}
```

Taking `last` of this would give the sum of the elements of s . To get a sequence whose elements are reversed from those of s , write

⁴ In the sequence form, `IN s` is not a set type but a special construct; treating it as a set type would throw away the essential ordering information.

```
{x :IN s, rev := {} BY {x} + rev}.last
```

To get the sequence $\{e, f(e), f^2(e), \dots, f^n(e)\}$, write

```
{i :IN 1 .. n, iter := e BY f(iter)}
```

This uses the `..` operator; $i .. j$ is the sequence $\{i, i+1, \dots, j-1, j\}$. It's the empty sequence if $i > j$.

Combinations

A combination is a way to combine the elements of a sequence or set into a single value using an infix operator, which must be associative, must have an identity, and must be commutative if it is applied to a set. You write “operator : sequence or set”. Thus

```
+ : (SEQ String){"He", "l", "lo"} = "He" + "l" + "lo" = "Hello"
```

because `+` on sequences is concatenation, and

```
+ : {i :IN 1 .. 4 | i**2} = 1 + 4 + 9 + 16 = 30
```

Existential and universal quantifiers make it easy to describe properties without explaining how to test for them in a practical way. For instance, a predicate that is `true` iff the sequence `s` is sorted is

```
(ALL i :IN 1 .. s.size-1 | s(i-1) <= s(i))
```

This is a common idiom; read it as

```
“for all i in 1 .. s.size-1, s(i-1) <= s(i)”.
```

This could also be written

```
(ALL i :IN (s.dom - {0}) | s(i-1) <= s(i))
```

since `s.dom` is the domain of the function `s`, which is the non-negative integers $< s.size$. Or it could be written

```
(ALL i :IN s.dom | i > 0 ==> s(i-1) <= s(i))
```

Because a universal quantification is just the conjunction of its predicate for all the values of the bound variables, it is simply a combination using `/\` as the operator:

```
(ALL i | Predicate(i)) = /\ : {i | Predicate(i)}
```

Similarly, an existential quantification is just a similar disjunction, hence a combination using `\/` as the operator:

```
(EXISTS i | Predicate(i)) = \/ : {i | Predicate(i)}
```

`Spec` has the redundant `ALL` and `EXISTS` notations because they are familiar.

If you want to get your hands on a value that satisfies an existential quantifier, you can construct the set of such values and use the `choose` method to pick out one of them:

```
{i | Predicate(i)}.choose
```

This is deterministic: `choose` always returns the same value given the same set (a necessary property for it to be a function). It is undefined if the set is empty, which is the case in the example if no `i` satisfies `Predicate`.

The `VAR` command described in the next section on commands is another form of existential quantification that lets you get your hands on the value, but it is non-deterministic.

Methods

Methods are a convenient way of packaging up some functions with a type so that the functions can be applied to values of that type concisely and without mentioning the type itself. Look at the

definitions in section 9 of the *Spec Reference Manual*, which give methods for the built-in types `SEQ T`, `SET T`, and `T->U`. If `s` is a `SEQ T`, `s.head` is `Sequence[T].Head(s)`, which is just `s(0)` (which is undefined if `s` is empty). You can see that it's shorter to write `s.head`.⁵

You can define your own methods by using `WITH`. For instance, consider

```
TYPE Complex = [re: Real, im: Real] WITH {"+":Add, mag:=Mag}
```

`Add` and `Mag` are ordinary `Spec` functions that you must define, but you can now invoke them on a `c` which is `Complex` by writing `c + c'` and `c.mag`, which mean `Add(c, c')` and `Mag(c)`. You can use existing operator symbols or make up your own; see section 3 of the reference manual for lexical rules. You can also write `Complex.+"` and `Complex.mag` to denote the functions `Add` and `Mag`; this may be convenient if `Complex` was declared in a different module. Using `Add` as a method does not make it private, hidden, static, local, or anything funny like that.

When you nest `WITH` the methods pile up in the obvious way. Thus

```
TYPE MoreComplex = Complex WITH {"-":Sub, mag:=Mag2}
```

has an additional method `-`, the same `+` as `Complex`, and a different `mag`. Many people call this ‘inheritance’ and ‘overriding’.

Functions

A function is a set of ordered pairs; the first element of each pair comes from the functions *domain*, and the second from its *range*. A function produces at most one value for an argument; that is, two pairs can't have the same first element. A function may be partial, that is, undefined at some elements of its domain. The expression `f!x` is true if `f` is defined at `x`, false otherwise. Like everything (except types), functions are ordinary values in `Spec`.

Given a function, you can use a function constructor to make another one that is the same except at a particular argument, as in the `DB` example above. Another example is `f{x -> 0}`, which is the same as `f` except that it is 0 at `x`. If you have never seen a construction like this one, think about it for a minute. Suppose you had to implement it. If `f` is represented as a table of (argument, result) pairs, the code will be easy. If `f` is represented by code that computes the result, the code for the constructor is less obvious, but you can make a new piece of code that says

```
(\ y: Int | ( (y = x) => 0 [*] f(y) ))
```

Here `\` is ‘lambda’, and the subexpression `((y = x) => 0 [*] f(y))` is a conditional, modeled on the conditional commands we saw in the first section; its value is 0 if `y = x` and `f(y)` otherwise, so we have changed `f` just at 0, as desired. If the else clause `[*] f(y)` is omitted, the condition is undefined if `y # x`. Of course in a running program you probably wouldn't want to construct new functions very often, so a piece of `Spec` that is intended to be close to practical code must use function constructors carefully.

Functions can return functions as results. Thus `T->U->V` is the type of a function that takes a `T` and returns a function of type `U->V`, which in turn takes a `U` and returns a `V`. If `f` has this type, then `f(t)` has type `U->V`, and `f(t)(u)` has type `V`. Compare this with `(T, U)->V`, the type of a function which takes a `T` and a `U` and returns a `V`. If `g` has this type, `g(t)` doesn't type-check, and `g(t, u)` has type `V`. Obviously `f` and `g` are closely related, but they are not the same.

⁵ Of course, `s(0)` is shorter still, but that's an accident; there is no similar alternative for `s.tail`.

You can define your own functions either by lambda expressions like the one above, or more generally by function declarations like this one

```
FUNC NewF(y: Int) -> Int = RET ( (y = x) => 0 [*] f(y) )
```

The value of this `NewF` is the same as the value of the lambda expression. To avoid some redundancy in the language, the meaning of the function is defined by a command in which `RET` sub-commands specify the value of the function. The command might be syntactically non-deterministic (for instance, it might contain `VAR` or `[]`), but it must specify at most one result value for any argument value; if it specifies no result values for an argument or more than one value, the function is undefined there. If you need a full-blown command in a function constructor, you can write it with `LAMBDA` instead of `\`:

```
(LAMBDA (y: Int) -> Int = RET ( (y = x) => 0 [*] f(y) ))
```

You can *compose* two functions with the `*` operator, writing `f * g`. This means to apply `f` first and then `g`, so you read it “`f` then `g`”. It is often useful when `f` is a sequence (remember that a `SEQ T` is a function from $\{0, 1, \dots, \text{size}-1\}$ to `T`), since the result is a sequence with every element of `f` mapped by `g`. This is Lisp’s or Scheme’s “`map`”. So:

```
(0 .. 4) * {\ i: Int | i*i} = (SEQ Int){0, 1, 4, 9, 16}
```

since $0 .. 4 = \{0, 1, 2, 3, 4\}$ because `Int` has a method `..` with the obvious meaning: $i .. j = \{i, i+1, \dots, j-1, j\}$. In the section on constructors we saw another way to write

```
(0 .. 4) * {\ i: Int | i*i},
```

as

```
{i :IN 0 .. 4 | | i*i}.
```

This is more convenient when the mapping function is defined by an expression, as it is here, but it’s less convenient if the mapping function already has a name. Then it’s shorter and clearer to write

```
(0 .. 4) * factorial
```

rather than

```
{i :IN 0 .. 4 | | factorial(i)}.
```

A function `f` has methods `f.dom` and `f.rng` that yield its domain and range sets, `f.inv` that yields its inverse (which is undefined at `y` if `f` maps zero or more than one argument to `y`), and `f.rel` that turns it into a relation (see below). `f.restrict(s)` is the same as `f` on elements of `s` and undefined elsewhere. The *overlay* operator combines two functions, giving preference to the second: $(f1 + f2)(x)$ is $f2(x)$ if that is defined and $f1(x)$ otherwise. So $f\{3 \rightarrow 24\} = f + \{3 \rightarrow 24\}$.

If type `U` has method `m`, then the function type $v = T \rightarrow U$ has a “lifted” method `m` that composes `U.m` with `v`, unless `v` already has a `m` method. `v.m` is defined by

```
(\ v | (\ t | v(t).m))
```

so that $v.m = v * U.m$. For example, $\{\text{"a"}, \text{"ab"}, \text{"b"}\}.size = \{1, 2, 1\}$. If `m` takes a second argument of type `w`, then `v.m` takes a second argument of type $vw = T \rightarrow w$ and is defined pairwise on the intersection of the domains, so if $Q = SEQ Int, Q\{1, 2, 3\} + Q\{10, 20\} = Q\{11, 22\}$.

Relations

A relation `r` is a generalization of a function: an arbitrary set of ordered pairs, defined by a total function from pairs to `Bool`. Thus `r` can relate an element of its domain to any number of elements of its range (including none). Like a function, `r` has `dom`, `rng`, and `inv` methods (the inverse is obtained just by flipping the ordered pairs), and you can compose relations with `*`. The advantage of relations is simplicity and generality; for example, there’s no notion of “undefined” for relations. The drawback is that you can’t write `r(x)` (although you can write $\{x\} ** r$ for the set of values related to `x` by `r`; see below).

A relation `r` has methods

`r.func` to turn it into a function: `r.func(x)` is undefined unless `r` relates `x` to exactly one value.

`r.setF` to turn it into a set function: `r.setF(x)` is the set of elements that `r` relates to `x`. This is total. The inverse of `setF` is the `setRel` method for a function whose values are sets: `r.setF.setRel = r`, and `f.setRel.setF = f` if `f` yields sets.

If `s` is a set, `s.rel` is a relation that relates `nil` to each member of the set. The relation’s `rng` method inverts this. You can extend composition to sets in two ways:

`s * f` is the set that results from applying `f` to every member of `s`, just like $q * f$ for a sequence; it’s undefined if `f` is undefined on any member of `s`.

`s ** r` takes the view of `s` as a relation more seriously; it is the set of all values that are related to a member of `s` by `r`: $(s.rel * r).rng$, which is also $\bigcup : (s * r.setF)$. This is never undefined.

A method `m` of `U` is lifted to `SET U` and to relations to `U` just as it is to functions to `U`. If `m` takes a second argument, `R.m` works on all pairs of related values, so that `r.m(rr)` relates `t` to `u.m(w)` whenever `r` relates `t` to `u` and `rr` relates `t` to `w`. Thus if $S = SET Int, S\{1, 2\} + S\{10, 20\} = S\{11, 12, 21, 22\}$.

If `U` doesn’t have a method `m` but `Bool` does, then the lifting is done on the function that defines the relation, so that $r1 \setminus r2$ is the union of the relations, $r1 \wedge r2$ the intersection, $r1 - r2$ the difference, and $\sim r$ the complement.

Commands

Commands are for changing the state. `Spec` has a few simple commands, and seven operators for combining commands into bigger ones. The main simple commands are assignment and routine invocation. There are also simple commands to raise an exception, to return a function result, and to `SKIP`, that is, do nothing. If a simple command evaluates an undefined expression, it fails (see below).

You can write `i + := 3` instead of `i := i + 3`, and similarly with any other binary operator.

The operators on commands are:

- A conditional operator: `predicate => command`, read “if `predicate` then `command`”. The predicate is called a *guard*.
- Choice operators: `c1 [] c2` and `c1 [*] c2`, read ‘or’ and ‘else’.
- Sequencing operators: `c1 ; c2` and `c1 EXCEPT handler`. The handler is a special form of conditional command: `exception => command`.
- Variable introduction: `VAR id: T | command`, read “choose `id` of type `T` such that `command` doesn’t fail”.
- Loops: `DO command OD`.

Section 6 of the reference manual describes commands. *Atomic Semantics of Spec* gives a precise account of their semantics. It explains that the meaning of a command is a *relation* between a state and an outcome (a state plus an optional exception), that is, a set of possible state-to-outcome transitions.

Conditionals and choice

The figure below (copied from Nelson’s paper) illustrates conditionals and choice with some very simple examples. Here is how they work:

The command

```
p => c
```

means to do `c` if `p` is true. If `p` is false this command fails; in other words, it has no outcome. More precisely, if `s` is a state in which `p` is false or undefined, this command does not relate `s` to any outcome.

What good is such a command? One possibility is that `p` will be true some time in the future, and then the command will have an outcome and allow a transition. Of course this can only happen in a concurrent program, where there is something else going on that can make `p` true. Even if there’s no concurrency, there might be an alternative to this command. For instance, it might appear in the larger command

```
p => c
[] p' => c'
```

in which you read `[]` as ‘or’. This fails only if each of `p` and `p'` is false or undefined. If both are true (as in the 00 state in the south-west corner of the figure), it means to do either `c` or `c'`; the choice is non-deterministic. If `p'` is `~p` then they are never both false, and if `p` is defined this command is equivalent to

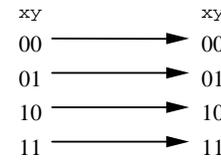
```
p => c
[*] c'
```

in which you read `[*]` as ‘else’. On the other hand, if `p` is undefined the two commands differ, because the first one fails (since neither guard can be evaluated), while the second does `c'`.

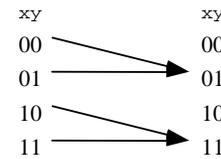
Both `c1 [] c2` and `c1 [*] c2` fail only if *both* `c1` and `c2` fail. If you think of a Spec program operationally (that is, as executing one command after another), this means that if the execution makes some choice that leads to failure later on, it must ‘back-track’ and try the other alternatives until it finds a set of choices that succeed. For instance, no matter what `x` is, after

```
y = 0 => x := x - 1; x < y => x := 1
[] y > 0 => x := 3 ; x < y => x := 2
[*] SKIP
```

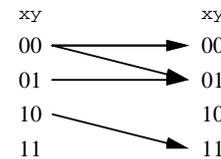
if `y = 0` initially, `x = 1` afterwards, if `y > 3` initially, `x = 2` afterwards, and otherwise `x` is unchanged. If you think of it relationally, `c1 [] c2` has all the transitions of `c1` (there are none if `c1` fails, several if it is non-deterministic) as well as all the transitions of `c2`. Both failure and non-determinism can arise from deep inside a complex command, not just from a top-level `[]` or `VAR`.



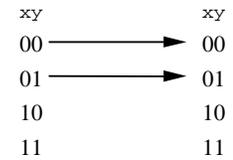
SKIP



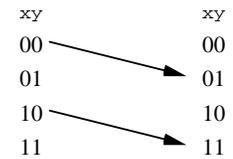
y := 1



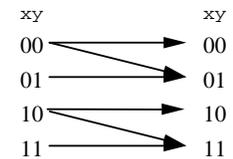
x = 0 => SKIP
[] y = 0 => y := 1
(partial, non-deterministic)



x = 0 => SKIP
(partial)



y = 0 => y := 1
(partial)



SKIP
[] y = 0 => y := 1
(non-deterministic)

Combining commands

The precedence rules for commands are

```
EXCEPT  binds tightest
;         next
=> |      next (for the right operand; the left side is an expression or delimited by VAR)
[] [*]    bind least tightly.
```

These rules minimize the need for parentheses, which are written around commands in the ugly form `BEGIN ... END` or the slightly prettier form `IF ... FI`; the two forms have the same meaning, but as a matter of style, the latter should only be used around guarded commands. So, for example,

```
p => c1; c2
```

is the same as

```
p => BEGIN c1; c2 END
```

and means to do `c1` followed by `c2` if `p` is true. To guard only `c1` with `p` you must write

```
IF p => c1 [*] SKIP FI; c2
```

which means to do `c1` if `p` is true, and then to do `c2`. The `[*] SKIP` ensures that the command before the `;` does not fail, which would prevent `c2` from getting done. Without the `[*] SKIP`, that is in

```
IF p => c1 FI; c2
```

if `p` is false the `IF ... FI` fails, so there is no possible outcome from which `c2` can be done and the whole thing fails. Thus `IF p => c1 FI; c2` has the same meaning as `p => BEGIN c1; c2 END`, which is a bit surprising.

Sequencing

A `c1 ; c2` command means just what you think it does: first `c1`, then `c2`. The command `c1 ; c2` gets you from state `s1` to state `s2` if there is an intermediate state `s` such that `c1` gets you from `s1` to `s` and `c2` gets you from `s` to `s2`. In other words, its relation is the composition of the relations for `c1` and `c2`; sometimes `;` is called ‘sequential composition’. If `c1` produces an exception, the composite command ignores `c2` and produces that exception.

A `c1 EXCEPT ex => c2` command is just like `c1 ; c2` except that it treats the exception `ex` the other way around: if `c1` produces the exception `ex` then it goes on to `c2`, but if `c1` produces a normal outcome (or any other exception), the composite command ignores `c2` and produces that outcome.

Variable introduction

`VAR` gives you more dramatic non-determinism than `[]`. The most common use is in the idiom

```
VAR x: T | P(x) => c
```

which is read “choose some `x` of type `T` such that `P(x)`, and do `c`”. It fails if there is no `x` for which `P(x)` is true and `c` succeeds. If you just write

```
VAR x: T | c
```

then `VAR` acts like ordinary variable declaration, giving an arbitrary initial value to `x`.

Variable introduction is an alternative to existential quantification that lets you get your hands on the bound variable. For instance, you can write

```
IF VAR n: Int, x: Int, y: Int, z: Int |
  (n > 2 /\ x**n + y**n = z**n) => out := n
[*] out := 0
FI
```

which is read: choose integers `n`, `x`, `y`, `z` such that `n > 2` and `xn + yn = zn`, and assign `n` to `out`; if there are no such integers, assign 0 to `out`.⁶ The command before the `[*]` succeeds iff $(\exists n: \text{Int}, x: \text{Int}, y: \text{Int}, z: \text{Int} \mid n > 2 \wedge x^n + y^n = z^n)$, but if we wrote that in a guard there would be no way to set `out` to one of the `n`’s that exist. We could also write

```
VAR s := { n: Int, x: Int, y: Int, z: Int
          | n > 2 /\ x**n + y**n = z**n
          | (n, x, y, z) }
```

to construct the set of all solutions to the equation. Then if `s # {}`, `s.choose` yields a tuple (n, x, y, z) with the desired property.

You can use `VAR` to describe all the transitions to a state that has an arbitrary relation `R` to the current state: `VAR s' | R(s, s') => s := s'` if there is only one state variable `s`.

The precedence of `|` is higher than `[]`, which means that you can string together different `VAR` commands with `[]` or `[*]`, but if you want several alternatives within a `VAR` you have to use `BEGIN ... END` or `IF ... FI`. Thus

```
VAR x: T | P(x) => c1
[] q => c2
```

is parsed the way it is indented and is the same as

```
BEGIN VAR x: T | P(x) => c1 END
[] BEGIN q => c2 END
```

but you must write the brackets in

```
VAR x: T |
  IF P(x) => c1
  [] Q(x) => c2
FI
```

which might be formatted more concisely as

```
VAR x: T |
  IF P(x) => c1
  [] R(x) => c2 FI
```

or even

```
VAR x: T | IF P(x) => c1 [] R(x) => c2 FI
```

You are supposed to indent your programs to make it clear how they are parsed.

Loops

You can always write a recursive routine, but often a loop is clearer. In Spec you use `DO ... OD` for this. These are brackets, and the command inside is repeated as long as it succeeds. When it fails, the repetition is over and the `DO ... OD` is complete. The most common form is

```
DO P => c OD
```

⁶ A correctness proof for an implementation of this spec defied the best efforts of mathematicians between Fermat’s time and 1993.

which is read “while P is true do c ”. After this command, P must be false. If the command inside the `DO ... OD` succeeds forever, the outcome is a looping exception that cannot be handled. Note that this is not the same as a failure, which simply means no outcome at all.

For example, you can zero all the elements of a sequence s with

```
VAR i := 0 | DO i < s.size => s(i) := 0; i - := 1 OD
```

or the simpler form (which also avoids fixing the order of the assignments)

```
DO VAR i | s(i) # 0 => s(i) := 0 OD
```

This is another common idiom: keep choosing an i as long as you can find one that satisfies some predicate. Since s is only defined for i between 0 and $s.size-1$, the guarded command fails for any other choice of i . The loop terminates, since the $s(i) := 0$ definitely reduces the number of i 's for which the guard is true. But although this is a good example of a loop, it is bad style; you should have used a sequence method or function composition:

```
s := S.fill(0, s.size)
```

or

```
s := {x :IN s | | 0}
```

(a sequence just like s except that every element is mapped to 0), remembering that Spec makes it easy to throw around big things. Don't write a loop when a constructor will do, because the loop is more complicated to think about. Even if you are writing code, you still shouldn't use a loop here, because it's quite clear how to write C code for the constructor.

To zero all the elements of s that satisfy some predicate P you can write

```
DO VAR i: Int | (s(i) # 0 /\ P(s(i))) => s(i) := 0 OD
```

Again, you can avoid the loop by using a sequence constructor and a conditional expression

```
s := {x :IN s | | (P(x) => 0 [*] x) }
```

Atomicity

Each `<<...>>` command is atomic. It defines a single transition, which includes moving the program counter (which is part of the state) from before to after the command. If a command is not inside `<<...>>`, it is atomic only if there's no reasonable way to split it up: `SKIP`, `HAVOC`, `RET`, `RAISE`. Here are the reasonable ways to split up the other commands:

- An assignment has one internal program counter value, between evaluating the right hand side expression and changing the left hand side variable.
- A guarded command likewise has one, between evaluating the predicate and the rest of the command.
- An invocation has one after evaluating the arguments and before the body of the routine, and another after the body of the routine and before the next transition of the invoking command.

Note that evaluating an expression is always atomic.

Modules and names

Spec's modules are very conventional. Mostly they are for organizing the name space of a large program into a two-level hierarchy: `module id`. It's good practice to declare everything except a few names of global significance inside a module. You can also declare `CONST`'s, just like `VAR`'s.

```
MODULE foo EXPORT i, j, Fact =
```

```
CONST c := 1
```

```
VAR i := 0
    j := 1
```

```
FUNC Fact(n: Int) -> Int =
    IF n <= 1 => RET 1
    [*] RET n * Fact(n - 1)
FI
```

```
END foo
```

You can declare an identifier `id` outside of a module, in which case you can refer to it as `id` everywhere; this is short for `Global.id`, so `Global` behaves much like an extra module. If you declare `id` at the top level in module m , `id` is short for `m.id` inside of m . If you include it in m 's `EXPORT` clause, you can refer to it as `m.id` everywhere. All these names are in the *global* state and are shared among all the atomic actions of the program. By contrast, names introduced by a declaration inside a routine are in the *local* state and are accessible only within their scope.

The purpose of the `EXPORT` clause is to define the external interface of a module. This is important because module T implements module S iff T 's behavior at its external interface is a subset of S 's behavior at its external interface.

The other feature of modules is that they can be parameterized by types in the same style as `CLU` clusters. The memory systems modules in handout 5 are examples of this.

You can also declare a class, which is a module that can be instantiated many times. The `Obj` class produces a global `Obj` type that has as its methods the exported identifiers of the class plus a `new` procedure that returns a new, initialized instance of the class. It also produces a `ObjMod` module that contains the declaration of the `Obj` type, the code for the methods, and a state variable indexed by `Obj` that holds the state records of the objects. For example:

```
CLASS Stat EXPORT add, mean, variance, reset =
```

```
VAR n          : Int := 0
    sum         : Int := 0
    sumsq       : Int := 0
```

```
PROC add(i: Int) = n + := 1; sum + := i; sumsq + := i**2
FUNC mean() -> Int = RET sum/n
FUNC variance() -> Int = RET sumsq/n - self.mean**2
PROC reset() = n := 0; sum := 0; sumsq := 0
```

```
END Stat
```

Then you can write

```
VAR s: Stat | s := s.new(); s.add(x); s.add(y); print(s.variance)
```

In abstraction functions and invariants we also write `obj.n` for field `n` in `obj`'s state.

Section 7 of the reference manual deals with modules. Section 8 summarizes all the uses of names and the scope rules. Section 9 gives several modules used to define the methods of the built-in data types such as functions, sets, and sequences.

This completes the language summary; for more details and greater precision consult the reference manual. The rest of this handout consists of three extended examples of specs and code written in Spec: topological sort, editor buffers, and a simple window system.

Example: Topological sort

Suppose we have a directed graph whose $n+1$ vertexes are labeled by the integers $0 \dots n$, represented in the standard way by a relation g ; $g(v_1, v_2)$ is true if v_2 is a successor of v_1 , that is, if there is an edge from v_1 to v_2 . We want a topological sort of the vertexes, that is, a sequence that is a permutation of $0 \dots n$ in which v_2 follows v_1 whenever v_2 is a successor of v_1 . Of course this possible only if the graph is acyclic.

```
MODULE TopologicalSort EXPORT V, G, Q, TopSort =
  TYPE V = IN 0 .. n                % Vertex
       G = (V, V) -> Bool          % Graph
       Q = SEQ V

  PROC TopSort(g) -> Q RAISES {cyclic} =
    IF VAR q | q IN (0 .. n).perms /\ IsTSorted(q, g) => RET q
    [*] RAISE cyclic                % g must be cyclic
    FI

  FUNC IsTSorted(q, g) -> Bool =
    % Not tsorted if v2 precedes v1 in q but is also a child
    RET ~ (EXISTS v1 :IN q.dom, v2 :IN q.dom | v2 < v1 /\ g(q(v1), q(v2)))

END TopologicalSort
```

Note that this solution checks for a cyclic graph. It allows any topologically sorted result that is a permutation of the vertexes, because the `VAR q` in `TopSort` allows any `q` that satisfies the two conditions. The `perms` method on sets and sequences is defined in section 9 of the reference manual; the `dom` method gives the domain of a function. `TopSort` is a procedure, not a function, because its result is non-deterministic; we discussed this point earlier when studying `SquareRoot`. Like that one, this spec has no internal state, since the module has no `VAR`. It doesn't need one, because it does all its work on the input argument.

The following code is from Cormen, Leiserson, and Rivest. It adds vertexes to the front of the output sequence as depth-first search returns from visiting them. Thus, a child is added before its parents and therefore appears after them in the result. Unvisited vertexes are `white`, nodes being visited are `grey`, and fully visited nodes are `black`. Note that all the descendants of a `black` node must be `black`. The `grey` state is used to detect cycles: visiting a `grey` node means that there is a cycle containing that node.

This module has state, but you can see that it's just for convenience in programming, since it is reset each time `TopSort` is called.

```
MODULE TopSortImpl EXPORT V, G, Q, TopSort = % implements TopSort
  TYPE Color = ENUM[white, grey, black] % plus the spec's types

  VAR out : Q
       color: V -> Color % every vertex starts white

  PROC TopSort(g) -> Q RAISES {cyclic} = VAR i := 0 |
    out := {}; color := {* -> white}
    DO VAR v | color(v) = white => Visit(v, g) OD; % visit every unvisited vertex
    RET out

  PROC Visit(v, g) RAISES {cyclic} =
    color(v) := grey;
    DO VAR v' | g(v, v') /\ color(v') # black => % pick an successor not done
      IF color(v') = white => Visit(v', g)
      [*] RAISE cyclic % grey — partly visited
      FI
    OD;
    color(v) := black; out := {v} + out % add v to front of out
```

The code is as non-deterministic as the spec: depending on the order in which `TopSort` chooses `v` and `Visit` chooses `v'`, any topologically sorted sequence can result. We could get deterministic code in many ways, for example by taking the smallest node in each case (the `min` method on sets is defined in section 9 of the reference manual):

```
VAR v := {v0 | color(v0) = white}.min in TopSort
VAR v' := {v0 | g(v, v0) /\ color(v') # black }.min in Visit
```

Code in C would do something like this; the details would depend on the representation of `G`.

Example: Editor buffers

A text editor usually has a *buffer* abstraction. A buffer is a mutable sequence of `c`'s. To get started, suppose that `C = Char` and a buffer has two operations,

```
Get(i) to get character i
```

```
Replace to replace a subsequence of the buffer by a subsequence of an argument of type SEQ C, where the subsequences are defined by starting position and size.
```

We can make this spec precise as a Spec class.

```

CLASS Buffer EXPORT B, C, X, Get, Replace =
TYPE X = Nat                                % index in buffer
    C = Char                                % Buffer contents
    B = SEQ C
VAR b : B := {}                             % Note: initially empty
FUNC Get(x) -> C = RET b(x)                 % Note: defined iff 0<=x<b.size
PROC Replace(from: X, size: X, b': B, from': X, size': X) =
% Note: fails if it touches C's that aren't there.
    VAR b1, b2, b3 | b = b1 + b2 + b3 /\ b1.size = from /\ b2.size = size =>
        b := b1 + b'.seg(from', size') + b3
END Buffer

```

We can implement a buffer as a sorted array of *pieces* called a ‘piece table’. Each piece contains a SEQ C, and the whole buffer is the concatenation of all the pieces. We use binary search to find a piece, so the cost of `Get` is at most logarithmic in the number of pieces. `Replace` may require inserting a piece in the piece table, so its cost is at most linear in the number of pieces.⁷ In particular, neither depends on the number of C’s. Also, each `Replace` increases the size of the array of pieces by at most two.

A piece is a B (in C it would be a pointer to a B) together with the sum of the length of all the previous pieces, that is, the index in `Buffer.b` of the first C that it represents; the index is there so that the binary search can work. There are internal routines `Locate(x)`, which uses binary search to find the piece containing `x`, and `Split(x)`, which returns the index of a piece that starts at `x`, if necessary creating it by splitting an existing piece. `Replace` calls `Split` twice to isolate the substring being removed, and then replaces it with a single piece. The time for `Replace` is linear in `pt.size` because on the average half of `pt` is moved when `Split` or `Replace` inserts a piece, and in half of `pt`, `p.x` is adjusted if `size' # size`.

```

CLASS BufImpl EXPORT B,C,X, Get, Replace = % implements Buffer
TYPE
    N = X % Types as in Buffer, plus
    P = [b, x] % iNdex in piece table
    PT = SEQ P % Piece: x is pos in Buffer.b
    % Piece Table
VAR pt := PT{}
ABSTRACTION FUNCTION buffer.b = + : {p :IN pt | | p.b}
% buffer.b is the concatenation of the contents of the pieces in pt
INVARIANT (ALL n :IN pt.dom | pt(n).b # {}
            /\ pt(n).x = + : {i :IN 0 .. n-1 | | pt(i).b.size})
% no pieces are empty, and x is the position of the piece in Buffer.b, as promised.
FUNC Get(x) -> C = VAR p := pt(Locate(x)) | RET p.b(x - p.x)
PROC Replace(from: X, size: X, b': B, from': X, size': X) =
    VAR n1 := Split(from); n2 := Split(from + size),
        new := P{b := b'.seg(from', size'), x := from} |
        pt := pt.sub(0, n1 - 1)
            + NonNull(new)
            + pt.sub(n2, pt.size - 1) * AdjustX(size' - size)
PROC Split(x) -> N =
% Make pt(n) start at x, so pt(Split(x)).x = x. Fails if x > b.size.
% If pt=abcd|efg|hi, then Split(4) is RET 1 and Split(5) is pt=abcd|e|fg|hi; RET 2
    IF pt = {} /\ x = 0 => RET 0
    [*] VAR n := Locate(x), p := pt(n), b1, b2 |
        p.b = b1 + b2 /\ p.x + b1.size = x =>
            VAR frag1 := p{b := b1}, frag2 := p{b := b2, x := x} |
                pt := pt.sub(0, n - 1)
                    + NonNull(frag1) + NonNull(frag2)
                    + pt.sub(n + 1, pt.size - 1);
                RET (b1 = {} => n [*] n + 1)
    FI
FUNC Locate(x) -> N = VAR n1 := 0, n2 := pt.size - 1 |
% Use binary search to find the piece containing x. Yields 0 if pt={},
% pt.size-1 if pt#{} /\ x>=b.size; never fails. The loop invariant is
% pt={ } \/ n2 >= n1 /\ pt(n1).x <= x /\ ( x < pt(n2).x \/ x >= pt.last.x )
% The loop terminates because n2 - n1 > 1 ==> n1 < n < n2, so n2 - n1 decreases.
    DO n2 - n1 > 1 =>
        VAR n := (n1 + n2)/2 | IF pt(n).x <= x => n1 := n [*] n2 := n FI
    OD; RET (x < pt(n2).x => n1 [*] n2)
FUNC NonNull(p) -> PT = RET (p.b # {} => PT{p} [*] {})
FUNC AdjustX(dx: Int) -> (P -> P) = RET (\ p | p{x + := dx})
END BufImpl

```

If subsequences were represented by their starting and ending positions, there would be lots of extreme cases to worry about.

⁷ By using a tree of pieces rather than an array, we could make the cost of `Replace` logarithmic as well, but to keep things simple we won’t do that. See `FSImpl` in handout 7 for more on this point.

Suppose we now want each `c` in the buffer to have not only a character code but also some additional properties, for instance the font, size, underlining, etc. `Get` and `Replace` remain the same. In addition, we need a third exported method `Apply` that applies to each character in a subsequence of the buffer a `map` function `C -> C`. Such a function might make all the `C`'s italic, for example, or increase the font size by 10%.

```
PROC Apply(map: C->C, from: X, size: X) =
  b := b.sub(0, from-1)
    + b.seg(from, size) * map
    + b.sub(from + size, b.size-1)
```

Here is code for `Apply` that takes time linear in the number of pieces. It works by changing the representation to add a `map` function to each piece, and in `Apply` composing the `map` argument with the `map` of each affected piece. We need a new version of `Get` that applies the proper `map` function, to go with the new representation.

```
TYPE P = [b, x, map: C->C] % x is pos in Buffer.b

ABSTRACTION FUNCTION buffer.b = + :{p :IN pt | | p.b * p.map}
% buffer.b is the concatenation of the pieces in p with their map's applied.
% This is the same AF we had before, except for the addition of * p.map.

FUNC Get(x) -> C = VAR p := pt(Locate(x)) | RET p.map(p.b(x - p.x))

PROC Apply(map: C->C, from: X, size: X) =
  VAR n1 := Split(from), n2 := Split(from + size) |
    pt := pt.sub(0, n1 - 1)
      + pt.sub(n1, n2 - 1) * (\ p | p{map := p.map * map})
      + pt.sub(n2, pt.size - 1)
```

Note that we wrote `Split` so that it keeps the same `map` in both parts of a split piece. We also need to add `map := (\ c | c)` to the constructor for `new` in `Replace`.

This code was used in the Bravo editor for the Alto, the first what-you-see-is-what-you-get editor. It is still used in Microsoft Word.

Example: Windows

A window (the kind on your computer screen, not the kind in your house) is a map from points to colors. There can be lots of windows on the screen; they are ordered, and closer ones block the view of more distant ones. Each window has its own coordinate system; when they are arranged on the screen, an offset says where each window's origin falls in screen coordinates.

```
MODULE Window EXPORT Get, Paint =
```

```
TYPE I = Int
  Coord = Nat
  Intensity = IN (0 .. 255).rng
  P = [x: Coord, y: Coord] WITH {"-":PSub} % Point
  C = [r: Intensity, g: Intensity, b: Intensity] % Color
  W = P -> C % Window

FUNC PSub(p1, p2) -> P = RET P{x := p1.x - p2.x, y := p1.y - p2.y}
```

The shape of the window is determined by the points where it is defined; obviously it need not be rectangular in this very general system. We have given a point a “-” method that computes the vector distance between two points; we somewhat confusingly represent the vector as a point.

A ‘window system’ consists of a sequence of `[w, offset: P]` pairs; we call a pair a `v`. The sequence defines the ordering of the windows (closer windows come first in the sequence); it is indexed by ‘window number’ `WN`. The `offset` gives the screen coordinate of the window's `(0, 0)` point, which we think of as its upper left corner. There are two main operations: `Paint(wn, p, c)` to set the value of `p` in window `wn`, and `Get(p)` to read the value of `p` in the topmost window where it is defined (that is, the first one in the sequence). The idea is that what you see (the result of `Get`) is the result of painting the windows from last to first, offsetting each one by its `offset` component and using the color that is painted later to completely overwrite one painted earlier. Of course real window systems have other operations to change the shape of windows, add, delete, and move them, change their order, and so forth, as well as ways for the window system to suggest that newly exposed parts of windows be repainted, but we won't consider any of these complications.

First we give the spec for a window system initialized with `n` empty windows. It is customary to call the coordinate system used by `Get` the screen coordinates. The `v.offset` field gives the screen coordinate that corresponds to `{0, 0}` in `v.w`. The `v.c(p)` method below gives the value of `v`'s window at the point corresponding to `p` after adjusting by `v`'s offset. The state `ws` is just the sequence of `v`'s. For simplicity we initialize them all with the same offset `{10, 5}`, which is not too realistic.

`Get` finds the smallest `WN` that is defined at `p` and uses that window's color at `p`. This corresponds to painting the windows from last (biggest `WN`) to first with opaque paint, which is what we wanted. `Paint` uses window rather than screen coordinates.

The state (the `VAR`) is a single sequence of windows.

```
TYPE WN = IN 0 .. n-1 % Window Number
  V = [w, offset: P] % window on the screen
    WITH {c:=(\ v, p | v.w(p - v.offset))} % C of a screen point p

VAR ws := {i :IN 0..n-1 | | V{{}, P{10,5}}} % the Window System

FUNC Get(p) -> C = VAR wn := {wn' | V.c!(ws(wn'), p)}.min | RET ws(wn).c(p)

PROC Paint(wn, p, c) = ws(wn).w(p) := c

END Window
```

Now we give code that only keeps track of the visible color of each point (that is, it just keeps the pixels on the screen, not all the pixels in windows that are covered up by other windows). We only keep enough state to handle `Get` and `Paint`.

The state is one `w` that represents the screen, plus an `exposed` variable that keeps track of which window is exposed at each point, and the offsets of the windows. This is sufficient to implement `Get` and `Paint`; to deal with erasing points from windows we would need to keep more information about what other windows are defined at each point, so that `exposed` would have a type `P -> SET WN`. Alternatively, we could keep track for each window of where it is defined.

Real window systems usually do this, and represent `exposed` as a set of visible regions of the various windows. They also usually have a ‘background’ window that covers the whole screen, so that every point on the screen has some color defined; we have omitted this detail from the spec and the code.

We need a history variable `wH` that contains the `w` part of all the windows. The abstraction function just combines `wH` and `offset` to make `ws`. The important properties of the code are contained in the invariant, from which it’s clear that `Get` returns the answer specified by `Window.Get`. Another way to do it is to have a history variable `wsH` that is equal to `ws`. This makes the abstraction function very simple, but then we need an invariant that says `offset(wn) = wsH(n).offset`. This is perfectly correct, but it’s usually better to put as little stuff in history variables as possible.

```
MODULE WinImpl EXPORT Get, Paint =
VAR w      := W{}                               % no points defined
    exposed : P -> WN := {}                       % which wn shows at p
    offset  := {i :IN 0..n-1 | P(5, 10)}          %
    wH      := {i :IN 0..n-1 | W{}}               % history variable

ABSTRACTION FUNCTION ws = (\ wn | V{w := wH(wn), offset := offset(wn)})

INVARIANT
  (ALL p | w!p = exposed!p
   /\ (w!p ==> {wn | V.c!(ws(wn), p)}.min = exposed(p)
   /\ w(p) = ws(exposed(p)).c(p) ) )
```

The invariant says that each visible point comes from some window, `exposed` tells the topmost window that defines it, and its color is the color of the point in that window. Note that for convenience the invariant uses the abstraction function; of course we could have avoided this by expanding it in line, but there is no reason to do so, since the abstraction function is a perfectly good function.

```
FUNC Get(p) -> C = RET w(p)

PROC Paint(wn, p, c) =
  VAR p0 | p = p0 - offset(wn) =>                % the screen coordinate
    IF wn <= exposed(p0) => w(p0) := c; exposed(p0) := wn [*] SKIP FI;
    wH(wn)(p) := c                               % update the history var

END WinImpl
```

Index

!, 9
 (. . .), 8
 . . . , 11
 ;, 14
 [*], 13
 [], 4, 13
 { * -> }, 8
 << . . . >>, 3
 ==>, 3, 10
 =>, 3, 12
 ->, 4, 10
 algorithm, 5
 ALL, 3, 10
 APROC, 4, 7
 arbitrary relation, 15
 array, 8
 assignment, 3, 8, 12
 atomic, 16
 atomic actions, 3
 atomic command, 6
 atomic procedure, 7
 BEGIN, 14
 behavior, 2
 Bool, 8
 choice, 12
 choose, 4, 10, 15
 class, 19
 client, 2
 code, 2
 combination, 10
 command, 3, 6, 12
 communicate, 2
 compose, 11
 composition, 16
 conditional, 11, 12
 constant, 7
 constructor, 8
 contract, 2
 declare, 7
 defined, 9
 Dijkstra, 1
 DO, 4, 16
 else, 14
 END, 14
 essential, 2
 EXCEPT, 14
 exception, 5
 exceptional outcome, 6
 existential quantifier, 5, 10
 expression, 4, 6
 fail, 12, 15
 FI, 14
 FUNC, 7
 function, 7, 8, 10
 function constructor, 8, 10
 function declaration, 11
 functional behavior, 2
 global, 17
 guard, 3, 12
 handler, 5
 hierarchy, 17
 history, 2, 6
 if, 3, 12
 IF, 14
 implementer, 2
 implication, 3
 infinite, 3
 Int, 8
 invocation, 12
 lambda expression, 8
 local, 3, 17
 loop, 16
 meaning
 of an atomic command, 6
 of an expression, 6
 method, 7, 11, 16
 module, 7, 17
 name, 6
 name space, 17
 Nelson, 1
 non-atomic command, 6
 non-atomic semantics, 6
 non-deterministic, 4, 5, 6, 13, 15
 normal outcome, 6, 14
 OD, 4, 16
 operator, 12
 or, 4, 13
 organizing your program, 7
 outcome, 6
 parameterized, 17
 precedence, 14, 15
 precisely, 2
 predicate, 3, 10, 12
 PROC, 7
 procedure, 7
 program, 2, 4, 7
 program counter, 6
 quantifier, 3, 4, 10
 RAISE, 5
 RAISES, 5
 record constructor, 8
 relation, 6
 repetition, 16
 RET, 4
 routine, 7
 seq, 11
 SEQ, 3
 sequence, 8, 16
 sequential program, 6
 set, 3, 8
 set constructor, 9
 set of sequences of states, 6
 side-effect, 7
 spec, 2
 specification, 2, 4
 state, 2, 6
 state transition, 2
 state variable, 6
 strongly typed, 8
 such that, 3
 SUCHTHAT, 8
 terminates, 16
 then, 3, 12
 thread, 7
 THREAD, 7
 transition, 2, 6
 two-level hierarchy, 7
 type, 7
 undefined, 8, 12
 universal quantifier, 3, 10
 value, 6
 VAR, 3, 4, 15
 variable, 6, 7
 variable introduction, 15
 WITH, 11

Operators (§ 5, § 9)

<i>Op</i>	<i>Pr</i>	<i>Type</i>	<i>x op y is</i>
.	9	Any	x 's y field/method
IS	8	Any	does x have type y ?
AS	8	Any	x with type y
**	8	Int	x^y
		set	compose with relation
*	7	Int	$x \times y$
		func	composition
		relation	composition
		set, seq	compose with function
/	7	Int	x/y rounded to 0
//	7	Int	mod: $x - (x/y)*y$
+	6	Int	$x + y$
		func	overlay
		seq	concatenation
-	6	Int	$x - y$
		set	set difference
		seq	multiset diff
!	6	func	x defined at y
!!	6	func	$x!y \wedge x(y)$ not ex
..	5	Int	seq $\{x, x+1, \dots, y\}$
=	4	Any	$x = y$
#	4	Any	$x \neq y$
==	4	seq	$x = y$ as multisets
<=	4	Int	$x \leq y$
		set	$x \subseteq y$ (subset)
		seq	x a prefix of y
<<=	4	seq	x a sub-seq of y
IN	4	set/seq	$x \in y$ (member)
~	3	Bool	not x (unary)
&\	2	Bool	$x \wedge y$ (and)
		set	$x \cap y$ (intersection)
&/	1	Bool	$x \vee y$ (or)
		set	$x \cup y$ (union)
==>	0	Bool	x implies y

Operators associate to the left.

Methods (§ 9)

set	<i>Ops:</i> * / \ \ / - <= IN, op:
size	number of members
choose	some member of s
seq	s as some sequence
rel	s as relation from nil
pred	$s.pred(x) = (x \in s)$
	(f)max/min a max/min by $f_1 / <=$
set/seq	perms set of all perms of sq
fsort	sq sorted (q stably) by f_1
sort	sq sorted (q stably) by $<=$
func	<i>Ops:</i> * + ! !!
dom, rng	domain, range
inv	inverse
restrict	domain to set s_1
rel	$r(x, y) = (f(x)=y)$
set	$s = \{x \mid pred(x)\}$
predicate	<i>Ops:</i> * and func +
relation	dom, rng domain, range
inv	inverse
setF	$f(x) = \{y \mid r(x, y)\}$
func	$f(x) = setF(x).choose$
graph	isPath is q_1 a path in g ?
closure	transitive closure of g
seq	<i>Ops:</i> + - .. <= <<= IN, op:, func * !
also see set/seq and func above	
size	number of elements
head	$q(0)$
tail	$\{q(1), \dots, q(q.size-1)\}$
remh	remove head = tail
last	$q(q.size-1)$
reml	$\{q(0), \dots, q(q.size-2)\}$
sub	$\{q(i_1), \dots, q(i_2)\}$
seg	$\{q(i_1), \dots\}, i_2$ elements
fill	i_2 copies of x_1
lexLE	q lexically $<= q_1$ by f_2 ?
fsorter	perm sorts q stably by f_1
count	number of x_1 's in q
tuple	tuple with q 's values
tuple	seq seq with tu 's values

Types (§ 4)

Any, Null, Bool, Int,	basic
Nat, Char, String	
SET T, IN s	set
$T_1 \rightarrow T_2$	function
APROC $T_1 \rightarrow T_2$	procedures
PROC $T_1 \rightarrow T_2$	
SEQ T	sequence
(T_1, \dots, T_n)	tuple
$[f_1: T_1, \dots, f_n: T_n]$	record
$(T_1 + \dots + T_n)$	union
T WITH $\{m_1 := f_1, \dots\}$	add methods
T SUCHTHAT pred	limit values

Commands (§ 6) *Pr*

SKIP, HAVOC,	simple
RET e, RAISE ex	
$p(e)$	invocation
$x := e, x := p(e),$ $(x_1, \dots) := e$	assignment
c_1 EXCEPT $ex \Rightarrow c_2$	3 handle ex
$c_1 ; c_2$	2 sequential
VAR $n: T \mid c$	1 new var n
pred => c	1 if (guarded cmd)
$c_1 [] c_2$	0 or (ND choice)
$c_1 [*] c_2$	0 else
<< c >>	atomic c
BEGIN c END	brackets
IF c FI	
DO c OD	loop until fail

Command operators associate to the left, but EXCEPT associates to the right.

Modules (§ 7)

MODULE/CLASS M	
$[T_1$ WITH $\{m_1: T_{11} \rightarrow T_{12}, \dots\}, \dots]$	
EXPORT $n_1, \dots =$	
TYPE $T_1 = SET T_2$	
$T_3 = ENUM[n_1, \dots]$	
CONST $n: T := e$	
VAR $n: T := e$	
EXCEPTION $ex = \{ex_{11}, \dots\} + ex_2 + \dots$	
FUNC $f(n_1: T_1, \dots) \rightarrow T = c$	
APROC, PROC, THREAD similarly	
END M	

Naming conventions (except in 'Operators')

c	command	op	operator
e	expression	p	procedure
ex	exception	Pr	precedence
f	function, field	q	sequence
g	graph	r	record, relation
i	Int	s	set
m	method	T	type
n	name	x	Any
z_i	i th extra argument of a method, or one of several like non-terminals in a rule		
§	a section of the Spec reference manual		

Expression forms (§ 5)

$f(e)$	func	function invocation
$op : sq$	set/seq	$sq(0)$ op $sq(1) \dots$
$(ALL \ x \mid pred)$	Bool	$pred(x_1) \wedge \dots \wedge pred(x_n)$
$(EXISTS \ x \mid pred)$	Bool	$pred(x_1) \vee \dots \vee pred(x_n)$
$(pred \Rightarrow e_1 [*] e_2)$	Any	e_1 if pred else e_2

Constructors (§ 5)

$\{e_1, \dots, e_n\}$	set	with these members
$\{i: Nat \mid i < 3 \mid i * 2\}$		of i^2 's where $i < 3$
$f\{e_1 \rightarrow e_2\}$	func	f except $= e_2$ at arg e_1
$\{* \rightarrow e\}$		$= e$ at every arg
$(\lambda i: Int \mid i < 3)$		lambda (also LAMBDA)
$\{e_1, \dots, e_n\}$	seq	of e 's in this order
$\{i: IN 0 .. 5 \mid i * 2\}$		$\{0, 1, 4, 9, 16, 25\}$
$\{i:=0 \text{ BY } i+1 \text{ WHILE } i < 6 \mid i * 2\}$		same
(e_1, \dots, e_n)	tuple	of e 's in this order
$r\{f_1 := e_1, \dots, f_n := e_n\}$	record	r except $f_1 = e_1 \dots$

How to Write a Spec

Figure out what the state is.

Choose the state to make the spec simple and clear, not to match the code.

Describe the actions.

What they do to the state.

What they return.

Helpful hints

Notation is important, because it helps you to think about what's going on.

Invent a suitable vocabulary.

Less is more. Less state is better. Fewer actions are better.

More non-determinism is better, because it allows more implementations.

In distributed systems, replace the separate nodes with non-determinism in the spec.

Pass the coffee-stain test: people should want to read the spec.

I'm sorry I wrote you such a long letter; I didn't have time to write a short one. — Pascal

How to Design an Implementation

Write the spec first.

Dream up the idea of the implementation.

Embody the key idea in the abstraction function.

Check that each implementation action simulates some spec actions.

Add invariants to make this easier. Each action must maintain them.

Change the implementation (or the spec, or the abstraction function) until this works.

Make the implementation correct first, then efficient.

More efficiency means more complicated invariants.

You might need to change the spec to get an efficient implementation.

Measure first before making anything faster.

An efficient program is an exercise in logical brinkmanship. — Dijkstra

4. Spec Reference Manual

Spec is a language for writing specifications and the first few stages of successive refinement towards practical code. As a specification language it includes constructs (quantifiers, backtracking or non-determinism, some uses of atomic brackets) which are impractical in final code; they are there because they make it easier to write clear, unambiguous and suitably general specs. If you want to write a practical program, avoid them.

This document defines the syntax of the language precisely and the semantics informally. **You should read the *Introduction to Spec* (handout 3) before trying to read this manual.** In fact, this manual is intended mainly for reference; rather than reading it carefully, skim through it, and then use the index to find what you need. For a precise definition of the atomic semantics read *Atomic Semantics of Spec*. (handout 9). Handout 17 on *Formal Concurrency* gives the non-atomic semantics semi-formally.

1. Overview

Spec is a notation for writing specs for a discrete system. What do we mean by a spec? It is the allowed sequences of transitions of a state machine. So Spec is a notation for describing sequences of transitions of a state machine.

Expressions and commands

The Spec language has two essential parts:

An *expression* describes how to compute a value as a function of other values, either constants or the current values of state variables.

A *command* describes possible transitions, or changes in the values of the state variables.

Both are based on the *state*, which in Spec is a mapping from names to values. The names are called state variables or simply variables: in the examples below they are i and j .

There are two kinds of commands:

An *atomic* command describes a set of possible transitions. For instance, the command $\langle\langle i := i + 1 \rangle\rangle$ describes the transitions $i=1 \rightarrow i=2$, $i=2 \rightarrow i=3$, etc. (Actually, many transitions are summarized by $i=1 \rightarrow i=2$, for instance, $(i=1, j=1) \rightarrow (i=2, j=1)$ and $(i=1, j=15) \rightarrow (i=2, j=15)$). If a command allows more than one transition from a given state we say it is *non-deterministic*. For instance, the command, $\langle\langle i := 1 \mid i := i + 1 \rangle\rangle$ allows the transitions $i=2 \rightarrow i=1$ and $i=2 \rightarrow i=3$. More on this in *Atomic Semantics of Spec*.

A *non-atomic* command describes a set of sequences of states. More on this in *Formal Concurrency*.

A sequential program, in which we are only interested in the initial and final states, can be described by an atomic command.

Spec's notation for commands, that is, for changing the state, is derived from Edsger Dijkstra's guarded commands (E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976) as extended by Greg Nelson (G. Nelson, A generalization of Dijkstra's calculus, *ACM TOPLAS* **11**, 4, Oct. 1989, pp 517-561). The notation for expressions is derived from mathematics.

Organizing a program

In addition to the expressions and commands that are the core of the language, Spec has four other mechanisms that are useful for organizing your program and making it easier to understand.

A *routine* is a named computation with parameters (passed by value). There are four kinds:

A *function* is an abstraction of an expression.

An *atomic procedure* is an abstraction of an atomic command.

A general procedure is an abstraction of a non-atomic command.

A *thread* is the way to introduce concurrency.

A *type* is a stylized assertion about the set of values that a name can assume. A type is also an easy way to group and name a collection of routines, called its *methods*, that operate on values in that set.

An *exception* is a way to report an unusual outcome.

A *module* is a way to structure the name space into a two-level hierarchy. An identifier i declared in a module m is known as i in m and as $m.i$ throughout the program. A *class* is a module that can be instantiated many times to create many objects.

A Spec program is some global declarations of variables, routines, types, and exceptions, plus a set of modules each of which declares some variables, routines, types, and exceptions.

Outline

This manual describes the language bottom-up:

- Lexical rules
- Types
- Expressions
- Commands
- Modules

At the end there are two sections with additional information:

- Scope rules
- Built-in methods for set, sequence, and routine types.

There is also an index. The *Introduction to Spec* has a one-page language summary.

2. Grammar rules

Nonterminal symbols are in lower case; terminal symbols are punctuation other than `::=`, or are quoted, or are in upper case.

Alternative choices for a nonterminal are on separate lines.

`symbol*` denotes zero or more occurrences of `symbol`.

The symbol `empty` denotes the empty string.

If `x` is a nonterminal, the nonterminal `xList` is defined by

```
xList ::= x
       x , xList
```

A comment in the grammar runs from `%` to the end of the line; this is just like Spec itself.

A `[n]` in a comment means that there is an explanation in a note labeled `[n]` that follows this chunk of grammar.

3. Lexical rules

The symbols of the language are literals, identifiers, keywords, operators, and the punctuation `() [] {} , ; : . | << >> := => -> [] [*]`. Symbols must not have embedded white space. They are always taken to be as long as possible.

A *literal* is a decimal number such as `3765`, a quoted character such as `'x'`, or a double-quoted string such as `"Hello\n"`.

An *identifier* (`id`) is a letter followed by any number of letters, underscores, and digits followed by any number of `'` characters. Case is significant in identifiers. By convention type and procedure identifiers begin with a capital letter. An identifier may not be the same as a keyword. The *predefined* identifiers `Any`, `Bool`, `Char`, `Int`, `Nat`, `Null`, `String`, `true`, `false`, and `nil` are declared in every program. The meaning of an identifier is established by a declaration; see section 8 on scope for details. Identifiers cannot be redeclared.

By convention *keywords* are written in upper case, but you can write them in lower case if you like; the same strings with mixed case are not keywords, however. The keywords are

ALL	APROC	AS	BEGIN	BY	CLASS
CONST	DO	END	ENUM	EXCEPT	EXCEPTION
EXISTS	EXPORT	FI	FUNC	HAVOC	IF
IN	IS	LAMBDA	MODULE	OD	PROC
RAISE	RAISES	RET	SEQ	SET	SKIP
SUCHTHAT	THREAD	TYPE	VAR	WHILE	WITH

An *operator* is any sequence of the characters `!@#$%^&*+==:.<>?/\|~` except the sequences `: . | << >> := => ->` (these are punctuation), or one of the keyword operators `AS`, `IN`, and `IS`.

A comment in a Spec program runs from a `%` outside of quotes to the end of the line. It does not change the meaning of the program.

4. Types

A type defines a set of values; we say that a value `v` has type `T` if `v` is in `T`'s set. The sets are not disjoint, so a value can belong to more than one set and therefore can have more than one type. In addition to its value set, a type also defines a set of routines (functions or procedures) called its *methods*; a method normally takes a value of the type as its first argument.

An expression has exactly one type, determined by the rules in section 5; the result of the expression has this type unless it is an exception.

The picky definitions given on the rest of this page are the basis for Spec's type-checking. You can skip them on first reading, or if you don't care about type-checking.

About unions: If the expression `e` has type `T` we say that `e` has a routine type `w` if `T` is a routine type `w` or if `T` is a union type and exactly one type `w` in the union is a routine type. Under corresponding conditions we say that `e` has a sequence or set type, or a record type with a field `f`.

Two types are *equal* if their definitions are the same (that is, have the same parse trees) after all type names have been replaced by their definitions and all `WITH` clauses have been discarded. Recursion is allowed; thus the expanded definitions might be infinite. Equal types define the same value set. Ideally the reverse would also be true, but type equality is meant to be decided by a type checker, whereas the set equality is intractable.

A type `T` *fits* a type `U` if the type-checker thinks they may have some values in common. This can only happen if they have the same structure, and each part of `T` fits the corresponding part of `U`. 'Fits' is an equivalence relation. Precisely, `T` fits `U` if:

`T = U`.

`T` is `T'` `SUCHTHAT F` or `(... + T' + ...)` and `T'` fits `U`, or vice versa. There may be no values in common, but the type-checker can't analyze the `SUCHTHAT` clauses to find out.

`T` and `U` are tuples of the same length and each component of `T` fits the corresponding component of `U`.

`T` and `U` are record types, and for every `decl id: T'` in `T` there is a corresponding `decl id: U'` in `U` such that `T'` fits `U'`, or vice versa.

`T=T1->T2 RAISES EXT` and `U=U1->U2 RAISES EXu`, or one or both `RAISES` are missing, and `T1` fits `U1` and `T2` fits `U2`. Similar rules apply for `PROC` and `APROC` types.

`T=SET T'` and `U=SET U'` and `T'` fits `U'`.

`T = Int->T'` or `SEQ T'` and `U = SEQ U'` and `T'` fits `U'`.

`T` *includes* `U` if the same conditions apply with "fits" replaced by "includes", all the "vice versa" clauses dropped, and in the `->` rule "`T1 fits U1`" replaced by "`U1 includes T1` and `EXT` is a superset of `EXu`". If `T` includes `U` then `T`'s value set includes `U`'s value set; again, the reverse is intractable.

An expression `e` fits a type `U` in state `s` if `e`'s type fits `U` and the result of `e` in state `s` has type `U` or is an exception; in general this can only be checked at runtime unless `U` includes `e`'s type. The check that `e` fits `T` is required for assignment and routine invocation; together with a few other checks it is called *type-checking*. The rules for type-checking are given in sections 5 and 6.

```

type ::= name % name of a type
      "Any" % every value has this type
      "Null" % with value set {nil}
      "Bool" % with value set {true, false}
      "Char" % like an enumeration
      "String" % = SEQ Char
      "Int" % integers
      "Nat" % naturals: non-negative integers
      SEQ type % sequence [1]
      SET type % set
      ( typeList ) % tuple; (T) is the same as T
      [ declList ] % record with declared fields
      ( union ) % union of the types
      aType -> type raises % function [2]
      APROC aType returns raises % atomic procedure
      PROC aType returns raises % non-atomic procedure
      type WITH { methodDefList } % attach methods to a type [3]
      type SUCHTHAT primary % restrict the value set [4]
      IN exp % = T SUCHTHAT ( \ t: T | t IN exp )
              % where exp's type has an IN method
      id [ typeList ] . id % type from a module [5]

name ::= id . id % the first id denotes a module
      id % short for m.id if id is declared
              % in the current module m, and for
              % Global.id if id is declared globally
      type . id % the id method of type
decl ::= id : type % id has this type
      id % short for id: Id [6]

union ::= type + type
       union + type

aType ::= ( )
       type

returns ::= empty % only for procedures
        -> type

raises ::= empty
        RAISES exceptionSet % the exceptions it can return

exceptionSet ::= { exceptionList } % a set of exceptions
              name % declared as an exception set
              exceptionSet \/ exceptionSet % set union
              exceptionSet - exceptionSet % set difference
exception ::= id % means "id"

method ::= id
        stringLiteral % the string must be an operator
              % other than "=" or "#" (see section 3)
methodDef ::= method := name % name is a routine

```

The ambiguity of the type grammar is resolved by taking \rightarrow to be right associative and giving WITH and RAISES higher precedence than \rightarrow .

[1] A SEQ T is just a function from $\{0, 1, \dots, \text{size}-1\}$ to T. That is, it is short for $(\text{Int} \rightarrow T)$ SUCHTHAT $(\ \backslash f: \text{Int} \rightarrow T \mid (\text{EXISTS size: Int} \mid (\text{ALL } i: \text{Int} \mid f!i = (i \text{ IN } 0 \dots \text{size}-1)))$ WITH { see section 9 }.

This means that invocation, !, and * work for a sequence just as they do for any function. In addition, there are many other useful operators on sequences; see section 9. The String type is just SEQ Char; there are String literals, defined in section 5.

[2] A T \rightarrow U value is a partial function from a state and a value of type T to a value of type U. A T \rightarrow U RAISES xs value is the same except that the function may raise the exceptions in xs.

[3] We say m is a *method* of T defined by f, and denote f by T.m, if

$T = T' \text{ WITH } \{ \dots, m := f, \dots \}$ and m is an identifier or is "op" where op is an operator (the construct in braces is a methodDefList), or

$T = T' \text{ WITH } \{ \text{methodDefList} \}$, m is not defined in methodDefList, and m is a method of T' defined by f, or

$T = (\dots + T' + \dots)$, m is a method of T' defined by f, and there is no other type in the union with a method m.

There are two special forms for invoking methods: $e1 \text{ infixOp } e2$ or $\text{prefixOp } e$, and $e1.\text{id}(e2)$ or $e.\text{id}$ or $e.\text{id}()$. They are explained in notes [1] and [3] to the expression grammar in the next section. This notation may be familiar from object-oriented languages. Unlike many such languages, Spec makes no provision for varying the method in each object, though it does allow inheritance and overriding.

A method doesn't have to be a routine, though the special forms won't type-check unless the method is a routine. Any method m of T can be referred to by T.m.

If type U has method m, then the function type $V = T \rightarrow U$ has a *lifted* method m that composes U.m with v, unless v already has a m method. $v.m$ is defined by $(\ \backslash v \mid (\ \backslash t \mid v(t).m))$

so that $v.m = v * U.m$. For example, $\{ "a", "ab", "b" \} . \text{size} = \{ 1, 2, 1 \}$. If m takes a second argument of type W, then $v.m$ takes a second argument of type $VW = T \rightarrow W$ and is defined on the intersection of the domains by applying m to the two results. Thus in this case $v.m$ is $(\ \backslash v, vv \mid (\ \backslash t : \text{IN } v.\text{dom} \ \backslash \ vv.\text{dom} \mid v(t).m(vv(t))))$

Lifting also works for relations to U, and therefore also for SET U. Thus if $R = (T, U) \rightarrow \text{Bool}$ and m returns type X, $R.m$ is defined by $(\ \backslash r \mid (\ \backslash t, x \mid x \text{ IN } \{ u \mid r(t, u) \mid u.m \}))$

so that $r.m = r * U.m.\text{rel}$. If m takes a second argument, then $R.m$ takes a second argument of type $RR = T \rightarrow W$, and $r.m(rr)$ relates t to $u.m(w)$ whenever r relates t to u and rr relates t to w. In other words, $R.m$ is defined by $(\ \backslash r, rr \mid (\ \backslash t, x \mid x \text{ IN } \{ u, w \mid r(t, u) \ \backslash \ rr(t, w) \mid u.m(w) \}))$

If U doesn't have a method m but Bool does, then the lifting is done on the function that defines the relation, so that $r1 \ \backslash \ r2$ is the union of the relations, $r1 \ \backslash \ r2$ the intersection, $r1 - r2$ the difference, and $\sim r$ the complement.

[4] In $T \text{ SUCHTHAT } f$, f is a predicate on T 's, that is, a function ($T \rightarrow \text{Bool}$). The type $T \text{ SUCHTHAT } f$ has the same methods as T , and its value set is the values of T for which f is true. See section 5 for `primary`.

[5] If a type is defined by `m[typeList].id` and `m` is a parameterized module, the meaning is `m'.id` where `m'` is defined by `MODULE m' = m[typeList] END m'`. See section 7 for a full discussion of this kind of type.

[6] `Id` is the `id` of a type, obtained from `id` by dropping trailing ' characters and digits, and capitalizing the first letter or all the letters (it's an error if these capitalizations yield different identifiers that are both known at this point).

5. Expressions

An expression is a partial function from states to results; results are values or exceptions. That is, an expression computes a result for a given state. The state is a function from names to values. This state is supplied by the command containing the expression in a way explained later. The meaning of an expression (that is, the function it denotes) is defined informally in this section. The meanings of invocations and lambda function constructors are somewhat tricky, and the informal explanation here is supplemented by a formal account in *Atomic Semantics of Spec*. Because expressions don't have side effects, the order of evaluation of operands is irrelevant (but see [5] and [13]).

Every expression has a type. The result of the expression is a member of this type if it is not an exception. This property is guaranteed by the *type-checking* rules, which require an expression used as an argument, the right hand side of an assignment, or a routine result to fit the type of the formal, left hand side, or routine range (see section 4 for the definition of 'fit'). In addition, expressions appearing in certain contexts must have *suitable* types: in `e1(e2)`, `e1` must have a routine type; in `e1+e2`, `e1` must have a type with a "+" method, etc. These rules are given in detail in the rest of this section. A union type is suitable if exactly one of the members is suitable. Also, if T is suitable in some context, so are $T \text{ WITH } \{ \dots \}$ and $T \text{ SUCHTHAT } f$.

An expression can be a literal, a variable known in the scope that contains the expression, or a function invocation. The form of an expression determines both its type and its result in a state:

`literal` has the type and value of the literal.

`name` has the declared type of `name` and its value in the current state, `state("name")`. The form `T.m` (where T denotes a type) is also a name; it denotes the `m` method of T . Note that if `name` is `id` and `id` is declared in the current module `m`, then it is short for `m.id`.

`invocation f(e)`: f must have a function (not procedure) type $U \rightarrow T \text{ RAISES EX}$ or $U \rightarrow T$ (note that a sequence is a function), and `e` must fit U ; then `f(e)` has type T . In more detail, if f has result `rf` and `e` has type U' and result `re`, then U' must fit U (checked statically) and `re` must have type U (checked dynamically if U' involves a union or `SUCHTHAT`; if the dynamic check fails the result is a fatal error). Then `f(e)` has type T .

If either `rf` or `re` is undefined, so is `f(e)`. Otherwise, if either is an exception, that exception is the result of `f(e)`; if both are, `rf` is the result.

If both `rf` and `re` are normal, the result of `rf` at `re` can be:

A normal value, which becomes the result of `f(e)`.

An exception, which becomes the result of `f(e)`. If `rf` is defined by a function body that loops, the result is a special looping exception that you cannot handle.

Undefined, in which case `f(e)` is undefined and the command containing it fails (has no outcome) — failure is explained in section 6.

A function invocation in an expression never affects the state. If the result is an exception, the containing command has an exceptional outcome; for details see section 6.

The other forms of expressions (`e.id`, `constructors`, prefix and infix operators, combinations, and quantifications) are all syntactic sugar for function invocations, and their results are obtained by the rule used for invocations. There is a small exception for conditionals [5] and for the conditional logical operators `/\`, `\/,` and `==>` that are defined in terms of conditionals [13].

			<i>(precedence)</i>	<i>argument/result types</i>	<i>operation</i>
exp	::= primary				
	prefixOp exp	% [1]			
	exp infixOp exp	% [1]			
	infixOp : exp	% exp's elements combined by op [2]	infixOp ::= ** % (8)	(Int, Int)->Int	exponentiate
	exp IS type	% (EXISTS x: type exp = x)	* % (7)	(Int, Int)->Int	multiply
	exp AS type	% error unless (exp IS type) [14]	%	(T->U, U->V)->(T->V) [12]	function composition
			/ % (7)	(Int, Int)->Int	divide
primary	::= literal		// % (7)	(Int, Int)->Int	remainder
	name		+ % (6)	(Int, Int)->Int	add
	primary . id	% method invocation [3] or record field	%	(SEQ T, SEQ T)->SEQ T [12]	concatenation
	primary arguments	% function invocation	%	(T->U, T->U)->(T->U) [12]	function overlay
	constructor		- % (6)	(Int, Int)->Int	subtract
	(exp)		%	(SET T, SET T)->SET T [12]	set difference;
	(quantif declList pred)	% /\:{d p} for ALL, \/ for EXISTS [4]	%	(SEQ T, SEQ T)->SEQ T [12]	multiset difference
	(pred => exp ₁ [*] exp ₂)	% if pred then exp ₁ else exp ₂ [5]	! % (6)	(T->U, T)->Bool [12]	function is defined
	(pred => exp ₁)	% undefined if pred is false	!! % (6)	(T->U, T)->Bool [12]	func has normal value
literal	::= intLiteral	% sequence of decimal digits	.. % (5)	(Int, Int)->SEQ Int [12]	subrange
	charLiteral	% 'x', x a printing character	<= % (4)	(Int, Int)->Bool	less than or equal
	stringLiteral	% "xxx", with \ escapes as in C	%	(SET T, SET T)->Bool [12]	subset
arguments	::= (expList)	% the arg is the tuple (expList)	< % (4)	(SEQ T, SEQ T)->Bool [12]	prefix
	()		%	(T, T)->Bool, T with <=	less than
			> % (4)	e1<e2 = (e1<=e2 /\ e1#e2)	greater than
constructor	::= { }	% empty function/sequence/set [6]	%	(T, T)->Bool, T with <=	greater or equal
	{ expList }	% sequence/set constructor [6]	>= % (4)	e1>e2 = e2<e1	greater or equal
	(expList)	% tuple constructor	%	e1>=e2 = e2<=e1	equal
	name { }	% name denotes a func/seq/set type [6]	= % (4)	(Any, Any)->Bool [1]	equal
	name { expList }	% name denotes a seq/set/record type [6]	# % (4)	(Any, Any)->Bool	not equal
	primary { fieldDefList }	% record constructor [7]	%	e1#e2 = ~ (e1=e2)	
	primary { exp -> result }	% function or sequence constructor [8]	<<= % (4)	(SEQ T, SEQ T)->Bool [12]	non-contiguous sub-seq
	primary { * -> result }	% function constructor [8]	IN % (4)	(T, SET T)->Bool [12]	membership
	(LAMBDA signature = cmd)	% function with the local state [9]	/\ % (2)	(Bool, Bool)->Bool [13]	conditional and
	(\ declList exp)	% short for (LAMBDA(d)->T=RET exp) [9]	%	(SET T, SET T)->SET T [12]	intersection
	{ declList pred exp }	% set constructor [10]	\/ % (1)	(Bool, Bool)->Bool [13]	conditional or
	{ seqGenList pred exp }	% sequence constructor [11]	%	(SET T, SET T)->SET T [12]	union
			==> % (0)	(Bool, Bool)->Bool [13]	conditional implies
fieldDef	::= id := exp		op % (5)	not one of the above	[1]
result	::= empty	% the function is undefined	prefixOp ::= - % (6)	Int->Int	negation
	exp	% the function yields exp	~ % (3)	Bool->Bool	complement
	RAISE exception	% the function yields exception	op % (5)	not one of the above	[1]
seqGen	::= id := exp BY exp WHILE exp	% sequence generator [11]			
	id :IN exp				
pred	::= exp	% predicate, of type Bool			
quantif	::= ALL				
	EXISTS				

The ambiguity of the expression grammar is resolved by taking the `infixOps` to be left associative and using the indicated precedences for the `prefixOps` and `infixOps` (with 8 for `IS` and `AS` and 5 for `:` or any operator not listed); higher numbers correspond to tighter binding. The precedence is determined by the operator symbol and doesn't depend on the operand types.

[1] The meaning of `prefixOp e` is `T. "prefixOp" (e)`, where `T` is `e`'s type, and of `e1 infixOp e2` is `T1. "infixOp" (e1, e2)`, where `T1` is `e1`'s type. The built-in types `Int` (and `Nat` with the same operations), `Bool`, sequences, sets, and functions have the operations given in the grammar. Section 9 on built-in methods specifies the operators for built-in types other than `Int` and `Bool`. Special case: `e1 IN e2` means `T2. "IN" (e1, e2)`, where `T2` is `e2`'s type.

Note that the `=` operator does not require that the types of its arguments agree, since both are `Any`. Also, `=` and `#` cannot be overridden by `WITH`. To define your own abstract equality, use a different operator such as `=="`.

[2] The `exp` must have type `SEQ T` or `SET T`. The value is the elements of `exp` combined into a single value by `infixOp`, which must be associative and have an identity, and must also be commutative if `exp` is a set. Thus

$$+ : \{i: \text{Int} \mid 0 < i \wedge i < 5 \mid i^{**2}\} = 1 + 4 + 9 + 16 = 30,$$

and if `s` is a sequence of strings, `+` is the concatenation of the strings. For another example, see the definition of quantifications in [4]. Note that the entire set is evaluated; see [10].

[3] Methods can be invoked by dot notation.

The meaning of `e.id` or `e.id()` is `T.id(e)`, where `T` is `e`'s type.

The meaning of `e1.id(e2)` is `T.id(e1, e2)`, where `T` is `e1`'s type.

Section 9 on built-in methods gives the methods for built-in types other than `Int` and `Bool`.

[4] A quantification is a conjunction (if the quantifier is `ALL`) or disjunction (if it is `EXISTS`) of the `pred` with the `id`'s in the `declList` bound to every possible value (that is, every value in their types); see section 4 for `decl`. Precisely, $(\text{ALL } d \mid p) = \wedge : \{d \mid p\}$ and $(\text{EXISTS } d \mid p) = \vee : \{d \mid p\}$. All the expressions in these expansions are evaluated, unlike `e2` in the expressions `e1 \/\ e2` and `e1 \/\ e2` (see [10] and [13]).

[5] A conditional `(pred => e1 [*] e2)` is not exactly an invocation. If `pred` is true, the result is the result of `e1` even if `e2` is undefined or exceptional; if `pred` is false, the result is the result of `e2` even if `e1` is undefined or exceptional. If `pred` is undefined, so is the result; if `pred` raises an exception, that is the result. If `[*] e2` is omitted and `pred` is false, the result is undefined.

[6] In a constructor `{expList}` each `exp` must have the same type `T`, the type of the constructor is $(\text{SEQ } T + \text{SET } T)$, and its value is the sequence containing the values of the `exp`s in the given order, which can also be viewed as the set containing these values.

If `expList` is empty the type is the union of all function, sequence and set types, and the value is the empty sequence or set, or a function undefined everywhere. If desired, these constructors can be prefixed by a name denoting a suitable set or sequence type.

A constructor `T{e1, ..., en}`, where `T` is a record type `[f1: T1, ..., fn: Tn]`, is short for a record constructor (see [7]) `T{f1:=e1, ..., fn:=en}`.

[7] The `primary` must have a record type, and the constructor has the same type as its `primary` and denotes the same value except that the fields named in the `fieldDefList` have the given

values. Each value must fit the type declared for its `id` in the record type. The `primary` may also denote a record type, in which case any fields missing from the `fieldDefList` are given arbitrary (but deterministic) values. Thus if `R=[a: Int, b: Int]`, `R{a := 3, b := 4}` is a record of type `R` with `a=3` and `b=4`, and `R{a := 3, b := 4}{a := 5}` is a record of type `R` with `a=5` and `b=4`. If the record type is qualified by a `SUCHTHAT`, the fields get values that satisfy it, and the constructor is undefined if that's not possible.

[8] The `primary` must have a function or sequence type, and the constructor has the same type as its `primary` and denotes a value equal to the value denoted by the `primary` except that it maps the argument value given by `exp` (which must fit the domain type of the function or sequence) to `result` (which must fit the range type if it is an `exp`). For a function, if `result` is empty the constructed function is undefined at `exp`, and if `result` is `RAISE exception`, then `exception` must be in the `RAISES` set of `primary`'s type. For a sequence `result` must not be empty or `RAISE`, and `exp` must be in `primary.dom` or the constructor expression is undefined.

In the `*` form the `primary` must be a function type or a function, and the value of the constructor is a function whose result is `result` at every value of the function's domain type (the type on the left of the `->`). Thus if `F=(Int->Int)` and `f=F{*->0}`, then `f` is zero everywhere and `f{4->1}` is zero except at 4, where it is 1. If this value doesn't have the function type, the constructor is undefined; this can happen if the type has a `SUCHTHAT` clause. For example, the type can't be a sequence.

[9] A `LAMBDA` constructor is a statically scoped function definition. When it is invoked, the meaning of the body is determined by the local state when the `LAMBDA` was evaluated and the global state when it is invoked; this is ad-hoc but convenient. See section 7 for `signature` and section 6 for `cmd`. The `returns` in the `signature` may not be `empty`. Note that a function can't have side effects.

The form $(\backslash \text{ declList } \mid \text{ exp})$ is short for $(\text{LAMBDA } (\text{ declList}) \rightarrow T = \text{RET } \text{ exp})$, where `T` is the type of `exp`. See section 4 for `decl`.

[10] A set constructor `{ declList | pred | exp }` has type `SET T`, where `exp` has type `T` in the current state augmented by `declList`; see section 4 for `decl`. Its value is a set that contains `x` iff $(\text{EXISTS } \text{ declList } \mid \text{ pred } \wedge x = \text{ exp})$. Thus

$$\{i: \text{Int} \mid 0 < i \wedge i < 5 \mid i^{**2}\} = \{1, 4, 9, 16\}$$

and both have type `SET Int`. If `pred` is omitted it defaults to `true`. If `| exp` is omitted it defaults to the last `id` declared:

$$\{i: \text{Int} \mid 0 < i \wedge i < 5\} = \{1, 2, 3, 4\}$$

Note that if `s` is a set or sequence, `IN s` is a type (see section 4), so you can write a constructor like `{i :IN s | i > 4}` for the elements of `s` greater than 4. This is shorter and clearer than `{i | i IN s \/\ i > 4}`

If there are any values of the declared `id`'s for which `pred` is undefined, or `pred` is true and `exp` is undefined, then the result is undefined. If nothing is undefined, the same holds for exceptions; if more than one exception is raised, the result exception is an arbitrary choice among them.

[11] A sequence constructor `{ seqGenList | pred | exp }` has type `SEQ T`, where `exp` has type `T` in the current state augmented by `seqGenList`, as follows. The value of

$$\{x1 := e01 \text{ BY } e1 \text{ WHILE } p1, \dots, xn := e0n \text{ BY } en \text{ WHILE } pn \mid \text{ pred } \mid \text{ exp}\}$$

is the sequence which is the value of `result` produced by the following program. Here `exp` has type `T` and `result` is a fresh identifier (that is, one that doesn't appear elsewhere in the program). There's an informal explanation after the program.

```
VAR x2 := e02, ..., xn := e0n, result := T{ }, x1 := e01 |
DO p1 => x2 := e2; p2 => ... => xn := en; pn =>
  IF pred => result := result + {exp} [*] SKIP FI;
  x1 := e1
OD
```

However, `e0i` and `ei` are not allowed to refer to `xj` if $j > i$. Thus the `n` sequences are unrolled in parallel until one of them ends, as follows. All but the first are initialized; then the first is initialized and all the others computed, then all are computed repeatedly. In each iteration, once all the `xi` have been set, if `pred` is true the value of `exp` is appended to the result sequence; thus `pred` serves to filter the result. As with set constructors, an omitted `pred` defaults to `true`, and an omitted `| exp` defaults to `| xn`. An omitted `WHILE pi` defaults to `WHILE true`. An omitted `:= e0i` defaults to

```
:= {x: Ti | true}.choose
```

where `Ti` is the type of `ei`; that is, it defaults to an arbitrary value of the right type.

The generator `xi :IN ei` generates the elements of the sequence `ei` in order. It is short for

```
j := 0 BY j + 1 WHILE j < ei.size, xi BY ei(j)
```

where `j` is a fresh identifier. Note that if the `:IN` isn't the first generator then the first element of `ei` is skipped, which is probably not what you want. Note that `:IN` in a sequence constructor overrides the normal use of `IN s` as a type (see [10]).

Undefined and exceptional results are handled the same way as in set constructors.

Examples

<code>{i := 0 BY i+1 WHILE i <= n}</code>	<code>= 0..n = {0, 1, ..., n}</code>
<code>{r := head BY r.next WHILE r # nil r.val}</code>	the <code>val</code> fields of a list starting at <code>head</code>
<code>{x :IN s, sum := 0 BY sum + x}</code>	partial sums of <code>s</code>
<code>{x :IN s, sum := 0 BY sum + x}.last</code>	<code>+ : s</code> , the last partial sum
<code>{x :IN s, rev := {} BY {x} + rev}.last</code>	reverse of <code>s</code>
<code>{x :IN s f(x)}</code>	<code>s * f</code>
<code>{i :IN 1..n i // 2 # 0 i * i}</code>	squares of odd numbers $\leq n$
<code>{i :IN 1..n, iter := e BY f(iter)}</code>	<code>{f(e), f²(e), ..., fⁿ(e)}</code>

[12] These operations are defined in section 9.

[13] The conditional logical operators are defined in terms of conditionals:

```
e1 \ / e2 = ( e1 => true [*] e2 )
e1 /\ e2 = ( ~e1 => false [*] e2 )
e1 ==> e2 = ( ~e1 => true [*] e2 )
```

Thus the second operand is not evaluated if the value of the first one determines the result.

[14] `AS` changes only the type of the expression, not its value. Thus if `(exp IS type)` the value of `(exp AS type)` is the value of `exp`, but its type is `type` rather than the type of `exp`.

6. Commands

A command changes the state (or does nothing). Recall that the state is a mapping from names to values; we denote it by `state`. Commands are non-deterministic. An atomic command is one that is inside `<<...>>` brackets.

The meaning of an atomic command is a set of possible transitions (that is, a relation) between a state and an outcome (a state plus an optional exception); there can be any number of outcomes from a given state. One possibility is a looping exceptional outcome. Another is no outcomes. In this case we say that the atomic command *fails*; this happens because all possible choices within it encounter a false guard or an undefined invocation.

If a subcommand fails, an atomic command containing it may still succeed. This can happen because it's one operand of `[]` or `[*]` and the other operand succeeds. It can also happen because a non-deterministic construct in the language that might make a different choice. Leaving exceptions aside, the commands with this property are `[]` and `VAR` (because it chooses arbitrary values for the new variables). If we gave an operational semantics for atomic commands, this situation would correspond to backtracking. In the relational semantics that we actually give (in *Atomic Semantics of Spec*), it corresponds to the fact that the predicate defining the relation is the “or” of predicates for the subcommands. Look there for more discussion of this point.

A non-atomic command defines a collection of possible transitions, roughly one for each `<<...>>` command that is part of it. If it has simple commands not in atomic brackets, each one also defines a possible transition, except for assignments and invocations. An assignment defines two transitions, one to evaluate the right hand side, and the other to change the value of the left hand side. An invocation defines a transition for evaluating the arguments and doing the call and one for evaluating the result and doing the return, plus all the transitions of the body. These rules are somewhat arbitrary and their details are not very important, since you can always write separate commands to express more transitions, or atomic brackets to express fewer transitions. The motivation for the rules is to have as many transitions as possible, consistent with the idea that an expression is evaluated atomically.

A complete collection of possible transitions defines the possible sequences of states or histories; there can be any number of histories from a given state. A non-atomic command still makes choices, but it does not backtrack and therefore can have histories in which it gets stuck, even though in other histories a different choice allows it to run to completion. For the details, see handout 17 on formal concurrency.

cmd	::= SKIP	% [1]
	HAVOC	% [1]
	RET	% [2]
	RET exp	% [2]
	RAISE exception	% [9]
	CRASH	% [10]
	invocation	% [3]
	assignment	% [4]
	cmd [] cmd	% or [5]
	cmd [*] cmd	% else [5]
	pred => cmd	% guarded cmd: if pred then cmd [5]
	VAR declInitList cmd	% variable introduction [6]
	cmd ; cmd	% sequential composition
	cmd EXCEPT handler	% handle exception [9]
	<< cmd >>	% atomic brackets [7]
	BEGIN cmd END	% just brackets
	IF cmd FI	% just brackets [5]
	DO cmd OD	% repeat until cmd fails [8]
invocation	::= primary arguments	% primary has a routine type [3]
assignment	::= lhs := exp	% state := state{name -> exp} [4]
	lhs infixOp := exp	% short for lhs := lhs infixOp exp
	lhs := invocation	% of a PROC or APROC
	(lhsList) := exp	% exp a tuple that fits lhsList
	(lhsList) := invocation	
lhs	::= name	% defined in section 4
	lhs . id	% record field [4]
	lhs arguments	% function [4]
declInit	::= decl	% initially any value of the type [6]
	id : type := exp	% initially exp, which must fit type [6]
	id := exp	% short for id: T := exp, where
		% T is the type of exp
handler	::= exceptionSet => cmd	% [9]. See section 4 for exceptionSet

The ambiguity of the command grammar is resolved by taking the command composition operations `;`, `[]`, and `[*]` to be left-associative and `EXCEPT` to be right associative, and giving `[]` and `[*]` lowest precedence, `=>` and `| next` (to the right only, since their left operand is an `exp`), `;` next, and `EXCEPT` highest precedence.

[1] The empty command and `SKIP` make no change in the state. `HAVOC` produces an arbitrary outcome from any state; if you want to specify undefined behavior when a precondition is not satisfied, write `~precondition => HAVOC`.

[2] A `RET` may only appear in a routine body, and the `exp` must fit the result type of the routine. The `exp` is omitted iff the returns of the routine's signature is empty.

[3] For arguments see section 5. The argument are passed by value, that is, assigned to the formals of the procedure. A function body cannot invoke a `PROC` or `APROC`; together with the rule for assignments (see [7]) this ensures that it can't affect the state. An atomic command can invoke an `APROC` but not a `PROC`. A command is atomic iff it is `<< cmd >>`, a subcommand of an

atomic command, or one of the simple commands `SKIP`, `HAVOC`, `RET`, or `RAISE`. The type-checking rule for invocations is the same as for function invocations in expressions.

[4] You can only assign to a name declared with `VAR` or in a signature. In an assignment the `exp` must fit the type of the `lhs`, or there is a fatal error. In a function body assignments must be to names declared in the signature or the body, to ensure that the function can't have side effects.

An assignment to a left hand side that is not a name is short for assigning a constructor to a name. In particular,

```
lhs(arguments) := exp is short for lhs := lhs{arguments->exp}, and
lhs . id       := exp is short for lhs := lhs{id := exp}.
```

These abbreviations are expanded repeatedly until `lhs` is a name.

In an assignment the right hand side may be an invocation (of a procedure) as well as an ordinary expression (which can only invoke a function). The meaning of `lhs := exp` or `lhs := invocation` is to first evaluate the `exp` or do the invocation and assign the result to a temporary variable `v`, and then do `lhs := v`. Thus the assignment command is not atomic unless it is inside `<<...>>`.

If the left hand side of an assignment is a `(lhsList)`, the `exp` must be a tuple of the same length, and each component must fit the type of the corresponding `lhs`. Note that you cannot write a tuple constructor that contains procedure invocations.

[5] A guarded command fails if the result of `pred` is undefined or `false`. It is equivalent to `cmd` if the result of `pred` is `true`. A `pred` is just a Boolean `exp`; see section 4.

`S1 [] S2` chooses one of the `S1` to execute. It chooses one that doesn't fail. Usually `S1` and `S2` will be guarded. For example, `x=1 => y:=0 [] x> 1 => y:=1` sets `y` to 0 if `x=1`, to 1 if `x>1`, and has no outcome if `x<1`. But `x=1 => y:=0 [] x>=1 => y:=1` might set `y` to 0 or 1 if `x=1`.

`S1 [*] S2` is the same as `S1` unless `S1` fails, in which case it's the same as `S2`.

`IF ... FI` are just command brackets, but it often makes the program clearer to put them around a sequence of guarded commands, thus:

```
IF  x < 0 => y := 3
[]  x = 0 => y := 4
[*] y := 5
FI
```

[6] In a `VAR` the unadorned form of `declInit` initializes a new variable to an arbitrary value of the declared type. The `:=` form initializes a new variable to `exp`. Precisely,

```
VAR id: T := exp | c
```

is equivalent to

```
VAR id: T | id := exp; c
```

The `exp` could also be a procedure invocation, as in an assignment.

Several `declInits` after `VAR` is short for nested `VARS`. Precisely,

```
VAR declInit , declInitList | cmd
```

is short for

```
VAR declInit | VAR declInitList | cmd
```

This is unlike a module, where all the names are introduced in parallel.

[7] In an atomic command the atomic brackets can be used for grouping instead of `BEGIN ... END`; since the command can't be any more atomic, they have no other meaning in this context.

[8] Execute `cmd` repeatedly until it fails. If `cmd` never fails, the result is a looping exception that doesn't have a name and therefore can't be handled. Note that this is *not* the same as failure.

[9] Exception handling is as in Clu, but a bit simplified. Exceptions are named by literal strings (which are written without the enclosing quotes). A module can also declare an identifier that denotes a set of exceptions. A command can have an attached exception `handler`, which gets to look at any exceptions produced in the command (by `RAISE` or by an invocation) and not handled closer to the point of origin. If an exception is not handled in the body of a routine, it is raised by the routine's invocation.

An exception `ex` must be in the `RAISES` set of a routine `r` if either `RAISE ex` or an invocation of a routine with `ex` in its `RAISES` set occurs in the body of `r` outside the scope of a handler for `ex`.

[10] `CRASH` stops the execution of any current invocations in the module other than the one that executes the `CRASH`, and discards their local state. The same thing happens to any invocations outside the module from within it. After `CRASH`, no procedure in the module can be invoked from outside until the routine that invokes it returns. `CRASH` is meant to be invoked from within a special `Crash` procedure in the module that models the effects of a failure.

7. Modules

A program is some global declarations plus a set of modules. Each module contains variable, routine, exception, and type declarations.

Module definitions can be parameterized with `mformals` after the module `id`, and a parameterized module can be instantiated. Instantiation is like macro expansion: the formal parameters are replaced by the arguments throughout the body to yield the expanded body. The parameters must be types, and the body must type-check without any assumptions about the argument that replaces a formal other than the presence of a `WITH` clause that contains all the methods mentioned in the formal parameter list (that is, formals are treated as distinct from all other types).

Each module is a separate scope, and there is also a `Global` scope for the identifiers declared at the top level of the `program`. An identifier `id` declared at the top level of a non-parameterized module `m` is short for `m.id` when it occurs in `m`. If it appears in the `exports`, it can be denoted by `m.id` anywhere. When an identifier `id` that is declared globally occurs anywhere, it is short for `Global.id`. `Global` cannot be used as a module `id`.

An exported `id` must be declared in the module. If an exported `id` has a `WITH` clause, it must be declared in the module as a type with at least those methods, and only those methods are accessible outside the module; if there is no `WITH` clause, all its methods and constructors are accessible. This is Spec's version of data abstraction.

```

program      ::= toplevel* module* END
module       ::= modclass id mformals exports = body END id
modclass     ::= MODULE
               CLASS                               % [4]
exports      ::= EXPORT exportList
export       ::= id
               id WITH {methodList}                % see section 4 for method
mformals     ::= empty
               [ mfpList ]
mfp          ::= id
               id WITH { declList }                 % module formal parameter
                                                       % see section 4 for decl
body         ::= toplevel*
               id [ typeList ]                       % id must be the module id
                                                       % instance of parameterized module
toplevel     ::= VAR declInit*                       % declares the decl ids [1]
               CONST declInit*                      % declares the decl ids as constant
               routineDecl                          % declares the routine id
               EXCEPTION exSetDecl*                 % declares the exception set ids
               TYPE typeDecl*                       % declares the type ids and any
                                                       % ids in ENUMS
routineDecl  ::= FUNC id signature = cmd            % function
               APROC id signature = <<cmd>>         % atomic procedure
               PROC id signature = cmd              % non-atomic procedure
               THREAD id signature = cmd            % one thread for each possible
                                                       % invocation of the routine [2]
signature    ::= ( declList ) returns raises        % see section 4 for returns
               ( ) returns raises                   % and raises
exSetDecl    ::= id = exceptionSet                  % see section 4 for exceptionSet
typeDecl     ::= id = type                          % see section 4 for type
               id = ENUM [ idList ]                 % a value is one of the id's [3]

```

[1] The “`:= exp`” in a `declInit` (defined in section 6) specifies an initial value for the variable. The `exp` is evaluated in a state in which each variable used during the evaluation has been initialized, and the result must be a normal value, not an exception. The `exp` sees all the names known in the scope, not just the ones that textually precede it, but the relation “used during evaluation of initial values” on the variables must be a partial order so that initialization makes sense. As in an assignment, the `exp` may be a procedure invocation as well as an ordinary expression. It's a fatal error if the `exp` is undefined or the invocation fails.

[2] Instead of being invoked by the client of the module or by another procedure, a thread is automatically invoked in parallel once for every possible value of its arguments. The thread is named by the `id` in the declaration together with the argument values. So

```

VAR sum := 0, count := 0
THREAD P(i: Int) = i IN 0 .. 9 =>
  VAR t | t := F(i); <<sum := sum + t>>; <<count := count + 1>>

```

adds up the values of $F(0) \dots F(9)$ in parallel. It creates a thread $P(i)$ for every integer i ; the threads $P(0), \dots, P(9)$ for which the guard is true invoke $F(0), \dots, F(9)$ in parallel and total the results in `sum`. When `count = 10` the total is complete.

A thread is the only way to get an entire program to do anything (except evaluate initializing expressions, which could have side effects), since transitions only happen as part of some thread.

[3] The `id`'s in the list are declared in the module; their type is the `ENUM` type. There are no operations on enumeration values except the ones that apply to all types: equality, assignment, and routine argument and result communication.

[4] A class is shorthand for a module that declares a convenient object type. The next few paragraphs specify the shorthand, and the last one explains the intended usage.

If the class `id` is `Obj`, the module `id` is `ObjMod`. Each variable declared in a top level `VAR` in the class becomes a field of the `ObjRec` record type in the module. The module exports only a type `Obj` that is also declared globally. `Obj` indexes a collection of state records of type `ObjRec` stored in the module's `objs` variable, which is a function `Obj->ObjRec`. `Obj`'s methods are all the names declared at top level in the class except the variables, plus the `new` method described below; the exported `Obj`'s methods are all the ones that the class exports plus `new`.

To make a class routine suitable as a method, it needs access to an `ObjRec` that holds the state of the object. It gets this access through a `self` parameter of type `Obj`, which it uses to refer to the object state `objs(self)`. To carry out this scheme, each routine in the module, unless it appears in a `WITH` clause in the class, is 'objectified' by giving it an extra `self` parameter of type `Obj`. In addition, in a routine body every occurrence of a variable `v` declared at top level in the class is replaced by `objs(self).v` in the module, and every invocation of an objectified class routine gets `self` as an extra first parameter.

The module also gets a synthesized and objectified `StdNew` procedure that adds a state record to `objs`, initializes it from the class's variable initializations (rewritten like the routine bodies), and returns its `Obj` index; this procedure becomes the `new` method of `Obj` unless the class already has a `new` routine.

A class cannot declare a `THREAD`.

The effect of this transformation is that a variable `obj` of type `Obj` behaves like an object. The state of the object is `objs(obj)`. The invocation `obj.m` or `obj.m(x)` is short for `ObjMod.m(obj)` or `ObjMod.m(obj, x)` by the usual rule for methods, and it thus invokes the method `m`; in `m`'s body each occurrence of a class variable refers to the corresponding field in `obj`'s state. `obj.new()` returns a new and initialized `Obj` object. The following example shows how a class is transformed into a module.

```

CLASS Obj EXPORT T1, f, p, ... = MODULE ObjMod EXPORT Obj WITH {T1, f, p, new } =
TYPE T1 = ... WITH {add:=AddT}      TYPE T1 = ... WITH {add:=AddT}
CONST c := ...                      CONST c := ...

VAR v1:T1:=ei, v2:T2:=pi(v1), ...   TYPE ObjRec = [v1: T1, v2: T2, ...]
                                   Obj = Int WITH {T1, c, f:=f, p:=p,
                                   AddT:=AddT, ..., new:=StdNew}
VAR  objs: Obj -> ObjRec := {}

FUNC f(p1: RT1, ...) = ... v1 ...   FUNC f(self: Obj, p1: RT1, ...) = ... objs(self).v1 ...
PROC p(p2: RT2, ...) = ... v2 ...   PROC p(self: Obj, p2: RT2, ...) = ... objs(self).v2 ...
FUNC AddT(t1, t2) = ...              FUNC AddT(t1, t2) = ...           % in T1's WITH, so not objectified
...                                   ...
PROC StdNew(self: Obj) -> Obj =
  VAR obj: Obj | ~ obj IN objs.dom =>
    objs(obj) := ObjRec{};
    objs(obj).v1 := ei;
    objs(obj).v2 := pi(objs(obj).v1);
    ...;
  RET obj

END Obj                               END ObjMod

TYPE Obj = ObjMod.Obj

```

In abstraction functions and invariants we also write `obj.n` for field `n` in `obj`'s state, that is, for `ObjMod.objs(obj).n`.

8. Scope

The declaration of an identifier is known throughout the smallest scope in which the declaration appears (redeclaration is not allowed). This section summarizes how scopes work in Spec; terms defined before section 7 have pointers to their definitions. A scope is one of

- the whole program, in which just the predefined (section 3), module, and globally declared identifiers are declared;
- a module;
- the part of a routineDecl or LAMBDA expression (section 5) after the =;
- the part of a VAR declInit | cmd command after the | (section 6);
- the part of a constructor or quantification after the first | (section 5).
- a record type or methodDefList (section 4);

An identifier is declared by

- a module `id`, `mfp`, or `oplevel` (for types, exception sets, `ENUM` elements, and named routines),
- a `decl` in a record type (section 4), | constructor or quantification (section 5), `declInit` (section 6), routine signature, or `WITH` clause of a `mfp`, or
- a `methodDef` in the `WITH` clause of a type (section 4).

An identifier may not be declared in a scope where it is already known. An occurrence of an identifier `id` always refers to the declaration of `id` which is known at that point, except when `id` is being declared (precedes a `:`, the `=` of a `oplevel`, the `:=` of a record constructor, or the `:=` or `BY` in a `seqGen`), or follows a dot. There are four cases for dot:

`moduleId . id` — the `id` must be exported from the basic module `moduleId`, and this expression denotes the meaning of `id` in that module.

`record . id` — the `id` must be declared as a field of the record type, and this expression denotes that field of `record`. In an assignment's lhs see [7] in section 6 for the meaning.

`typeId . id` — the `typeId` denotes a type, `id` must be a method of this type, and this expression denotes that method.

`primary . id` — the `id` must be a method of `primary`'s type, and this expression, together with any following arguments, denotes an invocation of that method; see [2] in section 5 on expressions.

If `id` refers to an identifier declared by a `oplevel` in the current module `m`, it is short for `m.id`. If it refers to an identifier declared by a `oplevel` in the program, it is short for `Global.id`. Once these abbreviations have been expanded, every name in the state is either global (contains a dot and is declared in a `oplevel`), or local (does not contain a dot and is declared in some other way).

Exceptions look like identifiers, but they are actually string literals, written without the enclosing quotes for convenience. Therefore they do not have scope.

9. Built-in methods

Some of the type constructors have built-in methods, among them the operators defined in the expression grammar. The built-in methods for types other than `Int` and `Bool` are defined below. Note that these are not complete definitions of the types; they do not include the constructors.

Sets

A set has methods for

computing union, intersection, and set difference (lifted from `Bool`; see note 3 in section 4), and adding or removing an element, testing for membership and subset;

choosing (deterministically) a single element from a set, or a sequence with the same members, or a maximum or minimum element, and turning a set into its characteristic predicate (the inverse is the predicate's `set` method);

composing a set with a function or relation, and converting a set into a relation from `nil` to the members of the set (the inverse of this is just the range of the relation).

We define these operations with a module that represents a set by its characteristic predicate. Precisely, `SET T` behaves as though it were `Set[T].S`, where

```
MODULE Set[T] EXPORT S =
```

```
TYPE S = Any->Bool SUCHTHAT (\ s | (ALL any | s(any) ==> (any IS T)))
% Defined everywhere so that type inclusion will work; see section 4.
    WITH {"\":=Union, "\&":=Intersection, "-":=Difference,
          "IN":=In, "<=":=Subset, choose:=Choose, seq:=Seq,
          pred:=Pred, perms:=Perms, fsort:=FSort, sort:=Sort,
          fmax:=FMax, fmin:=FMin, max:=Max, min:=Min
          "***":=ComposeF, rel:=Rel, "***":=ComposeR }

FUNC Union(s1, s2)->S      = RET (\ t | s1(t) \\/ s2(t)) % s1 \\/ s2
FUNC Intersection(s1, s2)->S = RET (\ t | s1(t) /\ s2(t)) % s1 /\ s2
FUNC Difference(s1, s2)->S = RET (\ t | s1(t) /\ ~s2(t)) % s1 - s2
FUNC In(s, t)->Bool       = RET s(t) % t IN s
FUNC Subset(s1, s2)->Bool = RET (ALL t | s1(t) ==> s2(t)) % s1 <= s2
FUNC Size(s)->Int        = % s.size
    VAR t | s(t) => RET Size(s-{t}) + 1 [*] RET 0
FUNC Choose(s)->T        = VAR t | s(t) => RET t % s.choose
% Not really, since VAR makes a non-deterministic choice,
% but choose makes a deterministic one. It is undefined if s is empty.
FUNC Seq(s)->SEQ T       = % s.seq
% Defined only for finite sets. Note that Seq chooses a sequence deterministically.
    RET {q: SEQ T | q.rng = s /\ q.size = s.size}.choose
FUNC Pred(s)->(T->Bool) = RET s % s.pred
% s.pred is just s. We define pred for symmetry with seq, set, etc.

FUNC Perms(s)->SET SEQ T = RET s.seq.perms % s.perms
FUNC FSort(s, f: (T,T)->Bool)->S = RET s.seq.fsort(f) % s.fsort(f); f is compare
FUNC Sort(s)->S          = RET s.seq.sort % s.sort; only if T has <=
FUNC FMax(s, f: (T,T)->Bool)->T = RET s.fsort(f).last % s.fmax(f); a max under f
FUNC FMin(s, f: (T,T)->Bool)->T = RET s.fsort(f).head % s.fmin(f); a min under f
FUNC Max(s)->T           = RET s.fmax(T."<=") % s.max; only if T has <=
FUNC Min(s)->T           = RET s.fmin(T."<=") % s.min; only if T has <=
% Note that these functions are undefined if s is empty. If there are extremal
% elements not distinguished by f or "<=", they make an arbitrary deterministic choice.
FUNC ComposeF(s, f: T->U)->SET U = RET {t :IN s | f(t)} % s * f; image of s under f
% ComposeF by analogy with sequences, where ordinary function composition applies pointwise to the elements.
FUNC Rel(s) -> ((Null, T)->Bool) = RET (\ null, t | t IN s)
% s.rel relates nil to every element of s
FUNC ComposeR(s, r: (T, U)->Bool)->SET U = RET(s.rel*r).rng % s ** r; image of s under r
% ComposeR is relational composition: anything you can get to by r, starting with a member of s. We could have written
explicitly: {t :IN s, u | r(t, u) | u}, or as \/: (s * r.setF), or as (s.rel * r).rng

END Set
```

There are constructors `{}` for the empty set, `{e1, e2, ...}` for a set with specific elements, and `{declList | pred | exp}` for a set whose elements satisfy a predicate. These constructors are described in [6] and [10] of section 5. Note that `{t | p}.pred = (\ t | p)`, and similarly `(\ t | p).set = {t | p}`. A method on `T` is lifted to a method on `s`, unless the name conflicts with one of `s`'s methods, exactly like lifting on `s.rel`; see note 3 in section 4.

Functions

The function types `T->U` and `T->U RAISES XS` have methods for

composition, overlay, inverse, and restriction;

testing whether a function is defined at an argument and whether it produces a normal (non-exceptional) result at an argument, and for the domain and range;

converting a function to a relation (the inverse is the relation's `func` method) or a function that produces a set to a relation with each element of the set (`setRel`; the inverse is the relation's `setF` method).

In other words, they behave as though they were `Function[T, U].F`, where (making allowances for the fact that `xs` and `v` are pulled out of thin air):

```
MODULE Function[T, U] EXPORT F =
TYPE F = T->U RAISES XS WITH {"*":=Compose, "+":=Overlay,
                             inv:=Inverse, restrict:=Restrict,
                             "!=":=Defined, "!!":=Normal,
                             dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}

R = (T, U) -> Bool

FUNC Compose(f, g: U -> V) -> (T -> V) = RET (\ t | g(f(t)))
% Note that the order of the arguments is reversed from the usual mathematical convention.

FUNC Overlay(f1, f2) -> F = RET (\ t | (f2!t => f2(t) [*] f1(t)))
% (f1 + f2) is f2(x) if that is defined, otherwise f1(x)

FUNC Inverse(f) -> (U -> T) = RET f.rel.inv.func
FUNC Restrict(f, s: SET T) -> F = RET (\ t | (t IN s => f(t)))

FUNC Defined(f, t)->Bool =
  IF f(t)=f(t) => RET true [*] RET false FI EXCEPT XS => RET true

FUNC Normal(f, t)->Bool =
  IF f(t)=f(t) => RET true [*] RET false FI EXCEPT XS => RET false

FUNC Domain(f) -> SET T = RET {t | f!t}
FUNC Range (f) -> SET U = RET {t | f!!t | f(t)}

FUNC Rel(f) -> R = RET (\ t, u | f(t) = u)
FUNC SetRel(f) -> ((T, V)->Bool) = RET (\ t, v | (f!t ==> v IN f(t) [*] false) )
% if U = SET V, f.setRel relates each t in f.dom to each element of f(t).

END Function
```

Note that there are constructors `{}` for the function undefined everywhere, `T{* -> result}` for a function of type `T` whose value is `result` everywhere, and `f{exp -> result}` for a function which is the same as `f` except at `exp`, where its value is `result`. These constructors are described in [6] and [8] of section 5. There are also lambda constructors for defining a function by a computation, described in [9] of section 5. A method on `U` is lifted to a method on `F`, unless the name conflicts with a method of `F`; see note 3 in section 4.

A total function `T->Bool` is a predicate and has an additional method to compute the set of `T`'s that satisfy the predicate (the inverse is the set's `pred` method). In other words, a predicate behaves as though it were `Predicate[T].P`, where

```
MODULE Predicate[T] EXPORT P =
TYPE P = T -> Bool WITH {set:=Set}
FUNC Set(p) -> SET T = RET {t | p(t)}
END Predicate
```

A predicate with `T = (U, V)` is a relation and has additional methods to turn it into a function `U -> V` or a function `U -> SET V`, and to get its domain and range, invert it or compose it (overriding the methods for a function). In other words, it behaves as though it were `Relation[U, V].R`, where (allowing for the fact that `w` is pulled out of thin air in `Compose`):

```
MODULE Relation[U, V] EXPORT R =
TYPE R = (U, V) -> Bool WITH {func:=Func, setF:=SetFunc, dom:=Domain, rng :=Range,
                             inv:=Inverse, "*":=Compose}

FUNC Func(r) -> (U -> V) =
% Func(r) is defined at u iff r relates u to a single v.
  RET (\ u | (r.setF(u).size = 1 => r.setF(u).choose))
FUNC SetFunc(r) -> (U -> SET V) = RET (\ u | {v | r(u, v)})
% SetFunc(r) is defined everywhere, returning the set of V's related to u.

FUNC Domain(r) -> SET U = RET {u, v | r(u, v) | u}
FUNC Range (r) -> SET V = RET {u, v | r(u, v) | v}

FUNC Inverse(r) -> ((V, U) -> Bool) = RET (\ v, u | r(u, v))

FUNC Compose(r: R, s: (V, W)->Bool) -> (U, W)->Bool =
  RET (\ u, w | (EXISTS v | r(u, v) /\ s(v, w)) )

END Relation
```

A method on `v` is lifted to a method on `R`, unless there's a name conflict; see note 3 in section 4.

A relation with `U = V` is a graph and has additional methods to yield the sequences of `U`'s that are paths in the graph and to compute the transitive closure. In other words, it behaves as though it were `Graph[U].G`, where

```
MODULE Graph[T] EXPORT G =
TYPE G = (T, T) -> Bool WITH {paths:=Paths, closure:=TransitiveClosure }
P = SEQ T

FUNC Paths(g) -> SET P = RET {p | (ALL i :IN p.dom - {0} | g(p(i-1), p(i)))}
% Any p of size <= 1 is a path by this definition.
FUNC TransitiveClosure(g) -> G = RET (\ t1, t2 |
  (EXISTS p | p.size > 1 /\ p.head = t1 /\ p.last = t2 /\ p IN g.paths ))

END Graph
```

Sequences

A function is called a sequence if its domain is a finite set of consecutive `Int`'s starting at 0, that is, if it has type

```
Q = Int -> T SUCHTHAT (\ q | (EXISTS size: Int | q.dom = (0 .. size-1).rng))
```

We denote this type (with the methods defined below) by `SEQ T`. A sequence inherits the methods of the function (though it overrides +), and it also has methods for

- detaching or attaching the first or last element,
- extracting a segment of a sequence, concatenating two sequences, or finding the size,
- making a sequence with all elements the same
- making a sequence into a tuple (`rng` makes it into a set),
- testing for prefix or sub-sequence (not necessarily contiguous),

composing with a relation (SEQ T inherits composing with a function),
 lexical comparison, permuting, and sorting,
 treating a sequence as a multiset with operations to:
 count the number of times an element appears, test membership and multiset equality,
 take differences, and remove an element ("+" or "\/" is union and add1 adds an
 element).

All these operations are undefined if they use out-of-range subscripts, except that a sub-sequence
 is always defined regardless of the subscripts, by taking the largest number of elements allowed
 by the size of the sequence.

We define the sequence methods with a module. Precisely, SEQ T is Sequence[T].Q, where:

```
MODULE Sequence[T] EXPORTS Q =
  TYPE I      = Int
  Q          = (I -> T)
              SUCHTHAT (\ q | (ALL i | q!i = (0 <= i /\ i < q.size)))
              WITH { size:=Size, seg:=Seg, sub:=Sub, "+":=Concatenate,
                    head:=Head, tail:=Tail, addh:=AddHead, remh:=Tail,
                    last:=Last, reml:=RemoveLast, addl:=AddLast,
                    fill:=Fill, tuple:=Tuple,
                    "<=":=Prefix, "<<=":=SubSeq,
                    "***":=ComposeR, lexLE:=LexLE, perms:=Perms,
                    fsorter:=FSorter, fsort:=FSort, sort:=Sort,

                    % These methods treat a sequence as a multiset (or bag).
                    count:=Count, "IN":=In, "==":=EqElem,
                    "\/":=Concatenate, "-":=Diff, set:=Q.rng }

  FUNC Size(q)-> Int = RET q.dom.size
  FUNC Sub(q, i1, i2) -> Q =
  % q.sub(i1, i2); yields {q(i1),...,q(i2)}, or a shorter sequence if i1 < 0 or i2 >= q.size
  RET ({0, i1}.max .. {i2, q.size-1}.min) * q

  FUNC Seg(q, i, n: I) -> Q = RET q.sub(i, i+n-1) % q.seg(i,n); n T's from q(i)
  FUNC Concatenate(q1, q2) -> Q = VAR q | % q1 + q2
  q.sub(0, q1.size-1) = q1 /\ q.sub(q1.size, q.size-1) = q2 => RET q
  FUNC Head(q) -> T = RET q(0) % q.head; first element
  FUNC Tail(q) -> Q = % q.tail; all but first
  q.size > 0 => RET q.sub(1, q.size-1)
  FUNC AddHead(q, t) -> Q = RET {t} + q % q.addh(t)
  FUNC Last(q) -> T = RET q(q.size-1) % q.last; last element
  FUNC RemoveLast(q) -> Q = % q.reml; all but last
  q # {} => RET q.sub(0, q.size-2)
  FUNC AddLast(q, t) -> Q = RET q + {t} % q.addl(t)
  FUNC Fill(t, n: I) -> Q = RET {i :IN 0 .. n-1 | | t} % yields i copies of t
  FUNC Prefix(q1, q2) -> Bool = % q1 <= q2
  RET (EXISTS q | q1 + q = q2)
  FUNC SubSeq(q1, q2) -> Bool = % q1 <<= q2
  % Are q1's elements in q2 in the same order, not necessarily contiguously.
```

```
RET (EXISTS p: SET Int | p <= q2.dom /\ q1 = p.seq.sort * q2)
FUNC ComposeR(q, r: (T, U)->Bool) -> SEQ U = % q ** r
% Elements related to nothing are dropped. If an element is related to several things, they appear in arbitrary order.
RET + : (q * r.setF * (\s: SET U | s.seq))
FUNC LexLE(q1, q2, f: (T,T)->Bool) -> Bool = % q1.lexLE(q2, f); f is <=
% Is q1 lexically less than or equal to q2. True if q1 is a prefix of q2,
% or the first element in which q1 differs from q2 is less.
RET q1 <= q2
  \ / (EXISTS i :IN q1.dom /\ q2.dom | q1.sub(0, i-1) = q2.sub(0, i-1)
  /\ q1(i) # q2(i)) /\ f(q1(i), q2(i))
FUNC Perms(q)->SET Q = % q.perms
RET {q' | (ALL t | q.count(t) = q'.count(t))}
FUNC FSorter(q, f: (T,T)->Bool)->SEQ Int = % q.fsorter(f); f is <=
% The permutation that sorts q stably. Note: can't use min to define this, since min is defined using sort.
VAR ps := {p :IN q.dom.perms % all perms that sort q
  | (ALL i :IN (q.dom - {0}) | f((p*q)(i-1), (p*q)(i))) } |
VAR p0 :IN ps | % the one that reorders the least
  (ALL p :IN ps | p0.lexLE(p, Int."<=")) => RET p0
FUNC FSort(q, f: (T,T)->Bool) -> Q = % q.fsort(f); f is <= for the sort
RET q.fsorter(f) * q
FUNC Sort(q)->Q = RET q.fsort(T."<=") % q.sort; only if T has <=
FUNC Count(q, t)->Int = RET {t' :IN q | t' = t}.size % q.count(t)
FUNC In(t, q)->Bool = RET (q.count(t) # 0) % t IN q
FUNC EqElem(q1, q2) -> Bool = RET q1 IN q2.perms % q1 == q2; equal as multisets
FUNC Diff(q1, q2) -> Q = % q1 - q2
RET {q | (ALL t | q.count(t) = {q1.count(t) - q2.count(t), 0}.max)}.choose
END Sequence
```

We can't program Tuple in Spec, but it is defined as follows. If q: SEQ T, then q.tuple is a
 tuple of q.size T's, the first equal to q(0), the second equal to q(1), and so forth. For the
 inverse, if u is a tuple of T's, then u.seq is a SEQ T such that u.seq.tuple = u. If u is a tuple in
 which not all the elements have the same declared type, then u.seq is a SEQ Any such that
 u.seq.tuple = u.

Int has a method .. for making sequences: i .. j = {i, i+1, ..., j-1, j}. If j < i,
 i .. j = {}. You can also write i .. j as {k := i BY k + 1 WHILE k <= j}; see [11] in
 section 5. Int also has a seq method: i.seq = 0 .. i-1.

There is a constructor {e1, e2, ...} for a sequence with specific elements and a constructor
 {} for the empty sequence. There is also a constructor q{e1 -> e2}, which is equal to q except
 at e1 (and undefined if e1 is out of range). For the constructors see [6] and [8] of section 5. To
 generate a sequence there are constructors {x :IN q | pred | exp} and {x := e1 BY e2
 WHILE pred1 | pred2 | exp}. For these see [11] of section 5.

To map each element t of q to f(t) use function composition q * f. Thus if q: SEQ Int,
 q * (\ i: Int | i*i) yields a sequence of squares. You can also write this
 {i :IN q | | i*i}.

Index

- , 9, 19, 21
- !, 9, 22
- !! , 9, 22
- # , 9, 10
- %, 3
- (), 3, 8, 16
- (expList), 8
- (typeList), 5
- *, 3, 9, 21, 22
- ** , 9
- .. , 3, 5
- /. , 9
- // , 9
- /\ , 9, 20
- : , 8, 14
- :, 3, 5
- := , 3, 8, 14, 18
- ; , 14
- [] , 3, 14
- [declList], 5
- [*], 3, 8, 14
- [n], 3
- \ , 8
- \\ , 9, 20
- {* -> result}, 8
- { }, 3, 8
- {declList | pred | exp}, 8
- {exceptionList}, 5
- {exp -> result}, 8
- {expList}, 8
- {methodDefList}, 5, 6
- {}, 21
- {e1, e2, ...}, 21
- | , 3, 14
- ~, 9
- + , 5, 9, 19, 21, 22
- > , 9
- </<< , 14, 16
- << >> , 3
- <<= , 9, 22
- <= , 19, 21
- = , 9, 10
- ==> , 9
- => , 3, 8, 14
- > , 3, 5, 8
- > , 9
- <= /> , 9
- >> , 14
- abstract equality, 10
- add, 9
- add an element, 9
- adding an element, 19, 22
- add1, 22
- ALL, 8
- ambiguity, 10, 14
- Any, 5, 10
- append an element, 9
- APROC, 5, 16
- arguments, 8
- AS, 8
- assignment, 14
- associative, 6, 10, 14
- atomic command, 1, 13, 14
- atomic procedure, 2
- Atomic Semantics of Spec*, 1, 7, 13
- backtracking, 13
- bag, 22
- BEGIN, 14
- body, 16
- Bool, 5
- built-in methods, 19
- capital letter, 3
- Char, 5
- characteristic predicate, 19
- choice, 14
- choose, 19
- choosing an element, 19
- CLASS, 16
- class, 17
- closure, 21
- Clu, 15
- cmd, 14
- command, 1, 13
- comment, 3
- composition, 20
- concatenation, 9
- conditional, 10
- conditional and, 9
- conditional or, 9
- CONST, 17
- constructor, 8
- count, 22
- data abstraction, 6
- decl, 5
- declaration, 18
- defined, 9, 20
- difference, 22
- divide, 9
- DO, 14
- dot, 18
- e.id, 10
- e.id(), 10
- e1 infixOp e2, 10
- e1.id(e2), 10
- else, 14
- empty, 3, 10
- empty sequence, 22
- empty set, 19
- END, 14, 16
- ENUM, 16
- equal, 9
- equal types, 4
- EXCEPT, 14
- exception, 5, 6, 7, 15, 16
- exceptionSet, 5, 16
- EXISTS, 8
- exp, 8
- expanded definitions, 4
- expression, 1, 7
- expression has a type, 7
- fail, 7, 13
- FI, 14
- fill, 22
- fit, 4, 7, 11, 14, 15
- formal parameters, 16
- FUNC, 16, 23
- function, 2, 6, 14, 19, 20
- function undefined everywhere, 20
- general procedure, 2
- Global.id, 16, 18
- grammar, 3
- graph, 21
- greater or equal, 9
- greater than, 9
- grouping, 15
- guard, 13, 14
- handler, 14
- has a routine type, 4
- has type T, 4
- HAVOC, 14
- head, 22
- id, 3, 6
- id := exp, 8
- id [typeList], 5
- identifier, 3
- if, 14
- implies, 9
- IN, 9, 19, 22
- includes, 4
- infixOp, 9
- initial value, 17
- initialize, 15
- instantiate, 16
- intersection, 9, 19
- Introduction to Spec*, 1
- invocation, 7, 8, 10, 14
- IS, 8
- isEmpty, 22
- isPath, 21
- keyword, 3
- known, 20
- LAMBDA, 8, 11
- last, 19
- less than, 9
- lexical comparison, 22
- List, 3
- literal, 3, 7, 8
- local, 18
- logical operators, 12
- looping exception, 7, 13
- m[typeList].id, 6
- max, 19
- meaning
 - of an atomic command, 13
 - of an expression, 7
- membership, 9, 19
- method, 4, 5, 6, 19
- mfp, 16
- min, 19
- module, 2, 16
- multiply, 9
- multiset, 22
- multiset difference, 9
- name, 1, 5, 18
- new variable, 15
- non-atomic command, 2, 13
- Non-Atomic Semantics of Spec*, 1
- non-deterministic, 1
- nonterminal symbol, 3
- normal result, 20
- not equal, 9
- Null, 5
- OD, 14
- operator, 3, 6
- OrderedSet, 19
- organizing your program, 2
- outcome, 13
- parameterized module, 16
- path in the graph, 21
- precedence, 6, 9, 14
- precondition, 14
- pred, 8, 19
- predefined identifiers, 3
- predicate, 20
- prefix, 9, 22
- prefixOp, 9
- prefixOp e, 10
- primary, 8
- PROC, 5, 16
- program, 2, 16
- punctuation, 3
- quantif, 8
- quantification, 10
- quoted character, 3
- RAISE, 8, 14
- RAISE exception, 11
- RAISES, 5, 11
- RAISES set, 15
- record, 5, 10
- redeclaration, 18
- relation, 21
- remove an element, 9, 19, 22
- result, 7
- result type, 14
- RET, 14
- routine, 2, 14, 16
- scope, 18
- seq, 22
- SEQ, 5, 6, 22
- SEQ Char, 6
- sequence, 22
- sequential composition, 14
- sequential program, 2
- SET, 5, 10, 11, 19, 20
- set difference, 9, 19
- set of sequences of states, 2
- set of values, 4
- set with specific elements, 19
- setF, 21
- side effects, 15
- signature, 15, 16
- SKIP, 14
- specifications, 1
- state, 1, 7, 13, 18
- state machine, 1
- state variable, 1
- String, 5, 6
- stringLiteral, 5
- sub-sequence, 9, 22
- subset, 9, 19
- subtract, 9
- symbol, 3
- syntactic sugar, 7
- T.m, 6, 7
- T->U, 6
- tail, 22
- terminal symbol, 3
- test membership, 19, 22
- thread, 17
- toplevel, 16, 18
- totalF, 21
- transition, 1
- transitive closure, 21
- tuple, 5, 14, 15
- tuple constructor, 8
- type, 2, 4, 5, 16
- type equality, 4
- type inclusion, 4
- type-checking, 4, 7, 14
- undefined, 7, 10, 13
- UNION, 5, 6, 7, 9, 19, 22
- upper case, 3
- value, 1
- variable, 1, 14, 15
- white space, 3
- WITH, 5, 6, 10, 16

5. Examples of Specs and Code

This handout is a supplement for the first two lectures. It contains several example specs and code, all written using Spec.

Section 1 contains a spec for sorting a sequence. Section 2 contains two specs and one code for searching for an element in a sequence. Section 3 contains specs for a read/write memory. Sections 4 and 5 contain code for a read/write memory based on caching and hashing, respectively. Finally, Section 6 contains code based on replicated copies.

1. Sorting

The following spec describes the behavior required of a program that sorts sets of some type T with a " \leq " comparison method. We do not assume that " \leq " is antisymmetric; in other words, we can have $t_1 \leq t_2$ and $t_2 \leq t_1$ without having $t_1 = t_2$, so that " \leq " is not enough to distinguish values of T . For instance, T might be the record type `[name:String, salary: Int]` with " \leq " comparison of the `salary` field. Several T 's can have different names but the same `salary`.

```
APROC Sort(s: SET T) -> SEQ T = <<
  VAR q: SEQ T | (ALL i: T | s.count(i) = q.count(i)) /\ Sorted(b) => RET b >>
```

This spec uses the auxiliary function `Sorted`, defined as follows.

```
FUNC Sorted(q: SEQ T) -> Bool = RET (ALL i :IN q.dom - {0} | q(i-1) <= q(i))
```

If we made `Sort` a `FUNC` rather than a `PROC`, what would be wrong?¹ What could we change to make it a `FUNC`?

We could have written this more concisely as

```
APROC Sort(s: SET T) -> SEQ T =
  << VAR q :IN a.perms | Sorted(q) => RET q >>
```

using the `perms` method for sets that returns a set of sequences that contains all the possible permutations of the set.

2. Searching

Search spec

We begin with a spec for a procedure to search an array for a given element. Again, this is an `APROC` rather than a `FUNC` because there can be several allowable results for the same inputs.

```
APROC Search(q: SEQ T, x: T) -> Int RAISES {NotFound} =
  << IF VAR i: Int | (0 <= i /\ i < q.size /\ q(i) = x) => RET i
```

```
[*] RAISE NotFound
FI >>
```

Or, equivalently but slightly more concisely:

```
APROC Search(q: SEQ T, x: T) -> Int RAISES {NotFound} =
  << IF VAR i :IN q.dom | q(i) = x => RET i [*] RAISE NotFound FI >>
```

Sequential search code

Here is code for the `Search` spec given above. It uses sequential search, starting at the first element of the input sequence.

```
APROC SeqSearch(q: SEQ T, x: T) -> Int RAISES {NotFound} = << VAR i := 0 |
  DO i < q.size => IF q(i) = x => RET i [*] i + := 1 FI OD; RAISE NotFound >>
```

Alternative search spec

Some searching algorithms, for example, binary search, assume that the input argument sequence is sorted. Such algorithms require a different spec, one that expresses this requirement.

```
APROC Search1(q: SEQ T, x: T) -> Int RAISES {NotFound} = <<
  IF ~Sorted(q) => HAVOC
  [*] VAR i :IN q.dom | q(i) = x => RET i
  [*] RAISE NotFound
FI >>
```

You might consider writing the spec to raise an exception when the array is not sorted:

```
APROC Search2(q: SEQ T, x: T) -> Int RAISES {NotFound, NotSorted} = <<
  IF ~Sorted(q) => RAISE NotSorted
  ...
```

This is not a good idea. The whole point of binary search is to obtain $O(\log n)$ time performance (for a sorted input sequence). But any code for the `Search2` spec requires an $O(n)$ check, even for a sorted input sequence, in order to verify that the input sequence is in fact sorted.

This is a simple but instructive example of the difference between defensive programming and efficiency. If `Search` were part of an operating system interface, it would be intolerable to have `HAVOC` as a possible transition, because the operating system is not supposed to go off the deep end no matter how it is called (though it might be OK to return the wrong answer if the input isn't sorted; what would that spec be?). On the other hand, the efficiency of a program often depends on assumptions that one part of it makes about another, and it's appropriate to express such an assumption in a spec by saying that you get `HAVOC` if it is violated. We don't care to be more specific about what happens because we intend to ensure that it doesn't happen. Obviously a program written in this style will be more prone to undetected or obscure errors than one that checks the assumptions, as well as more efficient.

¹ Hint: a `FUNC` can't have side effects and must be deterministic (return the same value for the same arguments).

3. Read/write memory

The simplest form of read/write memory is a single read/write register, say of type v (for value), with arbitrary initial value. The following Spec module describes this:

```
MODULE Register [V] EXPORT Read, Write =
VAR x: V                                % arbitrary initial value
APROC Read() -> V = << RET x >>
APROC Write(v) = << x := v >>
END Register
```

Now we give a spec for a simple addressable memory with elements of type v . This is like a collection of read/write registers, one for each address in a set A . In other words, it's a function from addresses to data values. For variety, we include new `Reset` and `Swap` operations in addition to `Read` and `Write`.

```
MODULE Memory [A, V] EXPORT Read, Write, Reset, Swap =
TYPE M = A -> V
VAR m := Init()
APROC Init() -> M = << VAR m' | (ALL a | m'!a) => RET m' >>
% Choose an arbitrary function that is defined everywhere.
FUNC Read(a) -> V = << RET m(a) >>
APROC Write(a, v) = << m(a) := v >>
APROC Reset(v) = << m := M{* -> v} >>
% Set all memory locations to v.
APROC Swap(a, v) -> V = << VAR v' := m(a) | m(a) := v; RET v' >>
% Set location a to the input value and return the previous value.
END Memory
```

The next three sections describe code for `Memory`.

4. Write-back cache code

Our first code is based on two memory mappings, a main memory m and a *write-back cache* c . The code maintains the invariant that the number of addresses at which c is defined is constant. A real cache would probably maintain a weaker invariant, perhaps bounding the number of addresses at which c is defined.

```
MODULE WBCache [A, V] EXPORT Read, Write, Reset, Swap =
% implements Memory
TYPE M = A -> V
C = A -> V
CONST Csize : Int := ... % cache size
VAR m := InitM()
c := InitC()
APROC InitM() -> M = << VAR m' | (ALL a | m'!a) => RET m' >>
% Returns a M with arbitrary values.
APROC InitC() -> C = << VAR c' | c'.dom.size = Csize => RET c' >>
% Returns a C that has exactly Csize entries defined, with arbitrary values.
APROC Read(a) -> V = << Load(a); RET c(a) >>
APROC Write(a, v) = << IF ~c!a => FlushOne() [*] SKIP FI; c(a) := v >>
% Makes room in the cache if necessary, then writes to the cache.
APROC Reset(v) = <<...>> % exercise for the reader
APROC Swap(a, v) -> V = << VAR v' | Load(a); v' := c(a); c(a) := v; RET v' >>
% Internal procedures.
APROC Load(a) = << IF ~c!a => FlushOne(); c(a) := m(a) [*] SKIP FI >>
% Ensures that address a appears in the cache.
APROC FlushOne() =
% Removes one (arbitrary) address from the cache, writing the data value back to main memory if necessary.
<< VAR a | c!a => IF Dirty(a) => m(a) := c(a) [*] SKIP FI; c := c{a -> } >>
FUNC Dirty(a) -> Bool = RET c!a /\ c(a) # m(a)
% Returns true if the cache is more up-to-date than the main memory.
END WBCache
```

The following Spec function is an abstraction function mapping a state of the `WBCache` module to a state of the `Memory` module. It's written to live inside the module. It says that the contents of location a is $c(a)$ if a is in the cache, and $m(a)$ otherwise.

```
FUNC AF() -> M = RET (\ a | c!a => c(a) [*] m(a) )
```

5. Hash table code

Our second code for `Memory` uses a hash table for the representation.

```

MODULE HashMemory [A WITH {hf: A->Int}, V] EXPORT Read, Write, Reset, Swap =
% Implements Memory.
% The module expects that the hash function A.hf is total and that its range is 0 .. n for some n.

TYPE Pair      = [a, v]
  B            = SEQ Pair          % Bucket in hash table
  HashT       = SEQ B

VAR nb        := NumB()           % Number of Buckets
  m          := HashT.fill(B{}, nb) % Memory hash table; initially empty
  default    := V                % arbitrary default value

APROC Read(a) -> V = << VAR b := m(a.hf), i: Int |
  i := FindEntry(a, b) EXCEPT NotFound => RET default ; RET b(i).v >>

APROC Write(a, v) = << VAR b := DeleteEntry(a, m(a.hf)) |
  m(a.hf) := b + {Pair{a, v}} >>

APROC Reset(v) = << m := HashT.fill(B{}, nb); default := v >>

APROC Swap(a, v) -> V = << VAR v' | v' := Read(a); Write(a, v); RET v' >>

% Internal procedures.

FUNC NumBs() -> Int =
% Returns the number of buckets needed by the hash function; havoc if the hash function is not as expected.
  IF VAR n: Nat | A.hf.rng = (0 .. n).rng => RET n + 1 [*] HAVOC FI

APROC FindEntry(a, b) -> Int RAISES (NotFound) =
% If a appears in a pair in b, returns the index of some pair containing a; otherwise raises NotFound.
  << VAR i :IN b.dom | b(i).a = a => RET i [*] RAISE NotFound >>

APROC DeleteEntry(a, b) -> B << VAR i: Int |
% Removes some pair with address a from b, if any exists.
  i := FindEntry(a, b) EXCEPT NotFound => RET b ;
  RET b.sub(0, i-1) + b.sub(i+1, b.size-1) >>

END HashMemory

```

Note that `FindEntry` and `DeleteEntry` are `APROCS` because they are not deterministic when given arbitrary `b` arguments.

The following is a key invariant that holds between invocations of the operations of `HashMemory`:

```

FUNC Inv() -> Bool = RET
  ( nb > 0
  /\ m.size = nb
  /\ (ALL a | a.hf IN m.dom)
  /\ (ALL i :IN m.dom, p :IN m(i).rng | p.a.hf = i)
  /\ (ALL a | { j :IN m(a.hf).dom | m(a.hf)(j).a = a }.size <= 1) )

```

This says that the number of buckets is positive, that the hash function maps all addresses to actual buckets, that a pair containing address `a` appears only in the bucket at index `a.hf` in `m`, and

that at most one pair for an address appears in the bucket for that address. Note that these conditions imply that in any reachable state of `HashMemory`, each address appears in at most one pair in the entire memory.

The following `Spec` function is an abstraction function between states of the `HashMemory` module and states of the `Memory` module:

```

FUNC AF() -> M = RET
  (LAMBDA(a) -> V =
    IF VAR i :IN m.dom, p :IN m(i).rng | p.a = a => RET p.v
    [*] RET default
  FI)

```

That is, the data value for address `a` is any value associated with address `a` in the hash table; if there is none, the data value is the default value. `Spec` says that a function is undefined at an argument if its body can yield more than one result value. The invariants given above ensure that the `LAMBDA` is actually single-valued for all the reachable states of `HashMemory`.

Of course `HashMemory` is not fully detailed code. Its main deficiency is that it doesn't explain how to maintain the variable-length bucket sequences, which is usually done with a linked list. However, the code does capture all the essential details.

6. Replicated copies

Our final code is based on some number $k \geq 1$ of copies of each memory location. Initially, all copies have the same default value. A `Write` operation only modifies an arbitrary *majority* of the copies. A `Read` reads an arbitrary majority, and selects and returns the most recent of the values it sees. In order to allow the `Read` to determine which value is the most recent, each `Write` records not only its value, but also a sequence number.

For simplicity, we just show the module for a single read/write register. The constant `k` determines the number of copies.

```

MODULE MajorityRegister [V] = % implements Register

CONST k = 5

TYPE N = Nat
  Kint = IN 1 .. k % ints between 1 and k
  Maj = SET Kint % all majority subsets of Kint
        SUCHTHAT (\m: Maj | m.size > k/2)

TYPE P = [D, seqno: N] % Pair
  M = Kint -> P % Memory
  S = SET P

VAR default : D
  m := M[* -> P{d := default, seqno := 0}]

APROC Read() -> D = << RET ReadPair().d >>

APROC Write(d) = << VAR i: Int, maj |
% Determines the highest sequence number i, then writes d paired with i+1 to some majority maj of the copies.

```

```

i := ReadPair().seqno;
DO VAR j :IN maj | m(j).seqno # i+1 => m(j) := P{d := d, seqno := i+1} OD >>

```

% Internal procedures.

```

APROC ReadPair() -> P = << VAR s := ReadMaj () |
% Returns a pair with the largest sequence number from some majority of the copies.
VAR p :IN s | p.seqno = {p' :IN s | | p'.seqno}.max => RET p >>

APROC ReadMaj () -> S = << VAR maj | RET { i :IN maj | | m(i) } >>
% Returns the set of pairs belonging to some majority of the copies.

END MajorityRegister

```

We could have written the body of `ReadPair` as

```
<< VAR s := ReadMaj() | RET s.fmax((\ p1, p2 | p1.seqno <= p2.seqno)) >>
```

except that `fmax` always returns the same maximal `p` from the same `s`, whereas the `VAR` in `ReadPair` chooses one non-deterministically.

The following is a key invariant for `MajorityRegister`.

```

FUNC Inv(m: M) -> Bool = RET
  (ALL p :IN m.rng, p' :IN m.rng | p.seqno = p'.seqno ==> p.v = p'.v)
  /\ (EXISTS maj | (ALL i :IN maj, p :IN m.rng | m(i).seqno >= p.seqno))

```

The first conjunct says that any two pairs having the same sequence number also have the same data. The second conjunct says that the highest sequence number appears in some majority of the copies.

The following `Spec` function is an abstraction function between states of the `MajorityRegister` module and states of the `Register` module.

```
FUNC AF() -> V = RET m.rng.fmax((\ p1, p2 | p1.seqno <= p2.seqno)).v
```

That is, the abstract register data value is the data component of a copy with the highest sequence number. Again, because of the invariants, there is only one `p.v` that will be returned.

6. Abstraction Functions and Invariants

This handout describes the main techniques used to prove correctness of state machines: abstraction functions and invariant assertions. We demonstrate the use of these techniques for some of the `MEMORY` examples from handout 5.

Throughout this handout, we consider modules all of whose externally invocable procedures are `APROCS`. We assume that the body of each such procedure is executed all at once. Also, we do not consider procedures that modify global variables declared outside the module under consideration.

Modules as state machines

Our methods apply to an arbitrary state machine or automaton. In this course, however, we use a `Spec` module to define a state machine. Each state of the automaton designates values for all the variables declared in the module. The initial states of the automaton consist of initial values assigned to all the module's variables by the `Spec` code. The transitions of the automaton correspond to the invocations of `APROCS` together with their result values.

An *execution fragment* of a state machine is a sequence of the form $s_0, \pi_1, s_1, \pi_2, \dots$, where each s is a state, each π is a label for a transition (an invocation of a procedure), and each consecutive $(s_i, \pi_{i+1}, s_{i+1})$ triple follows the rules specified by the `Spec` code. (We do not define these rules here—wait for the lecture on atomic semantics.) An *execution* is an execution fragment that begins in an initial state.

The π_i are labels for the transitions; we often call them *actions*. When the state machine is written in `Spec`, each transition is generated by some atomic command, and we can use some unambiguous identification of the command for the action. At the moment we are studying sequential `Spec`, in which every transition is the invocation of an exported atomic procedure. We use the name of the procedure, the arguments, and the results as the label.

Figure 1 shows some of the states and transitions of the state machine for the `MEMORY` module with `A = IN 1 .. 4`, and Figure 2 does likewise for the `WBCache` module with `Csize = 2`. The arrow for each transition is labeled by its π_i , that is, by the procedure name, arguments, and result.

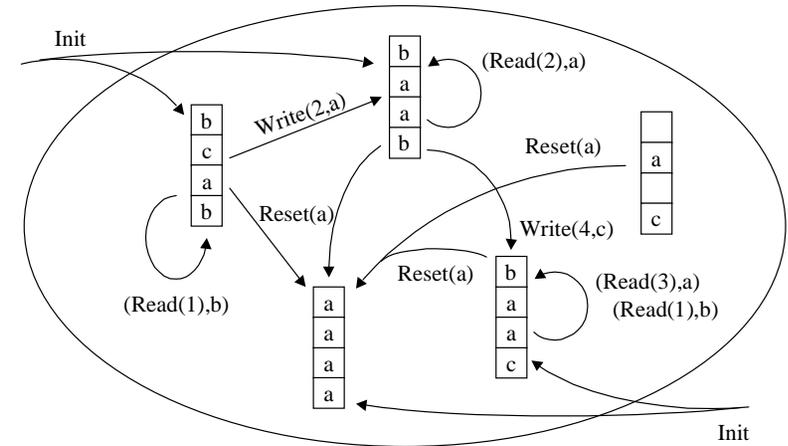


Figure 1: Part of the `MEMORY` state machine

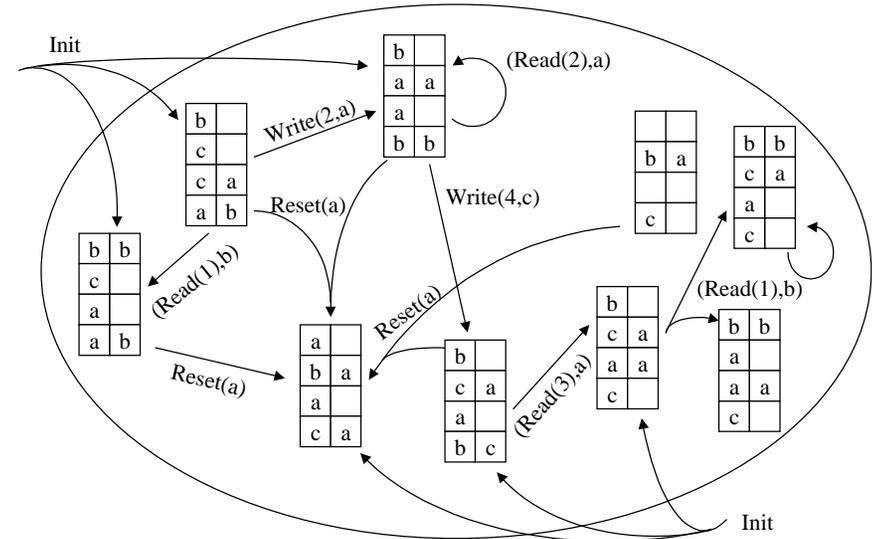


Figure 2: Part of the `WBCache` state machine

External behavior

Usually, a client of a module is not interested in all aspects of its execution, but only in some kind of external behavior. Here, we formalize the external behavior as a set of *traces*. That is, from an execution (or execution fragment) of a module, we discard both the states and the internal actions, and extract the *trace*. This is the sequence of labels π ; for external actions (that is, invocations of exported routines) that occur in the execution (or fragment). Then the external behavior of the module is the set of traces that are obtained from all of its executions.

It's important to realize that in going from the execution to the trace we are discarding a great deal of information. First, we discard all the states, keeping only the actions or labels. Second, we discard all the internal actions, keeping only the external ones. Thus the only information we keep in the trace is the behavior of the state machine at its external interface. This is appropriate, since we want to study state machines that have the same behavior at the external interface; we shall see shortly exactly what we mean by 'the same' here. Two machines can have the same traces even though they have very different state spaces.

In the sequential Spec that we are studying now, a module only makes a transition when an exported routine is invoked, so all the transitions appear in the trace. Later, however, we will introduce modules with internal transitions, and then the distinction between the executions and the external behavior will be important.

For example, the set of traces generated by the `Memory` module includes the following trace:

```
(Reset(v),)
(Read(a1),v)
(Write(a2,v'))
(Read(a2),v')
```

However, the following trace is not included if $v \neq v'$:

```
(Reset(v))
(Read(a1),v')           should have returned v
(Write(a2,v'))
(Read(a2),v)           should have returned v'
```

In general, a trace is included in the external behavior of `Memory` if every `Read(a)` or `Swap(a, v)` operation returns the last value written to `a` by a `Write`, `Reset` or `Swap` operation, or returned by a `Read` operation; if there is no such previous operation, then `Read(a)` or `Swap(a, v)` returns an arbitrary value.

Implements relation

In order to understand what it means for one state machine to implement another one, it is helpful to begin by considering what it means for one atomic procedure to implement another. The meaning of an atomic procedure is a relation between an initial state just before the procedure starts (sometimes called a 'pre-state') and a final state just after the procedure has finished (sometimes called a 'post-state'). This is often called an 'input-output relation'. For example, the relation defined by a square-root procedure is that the post-state is the same as the pre-state, except that the square of the procedure result is close enough to the argument. This meaning makes sense for an arbitrary atomic procedure, not just for one in a trace.

We say that procedure P implements spec S if the relation defined by P (considered as a set of ordered pairs of states) is a subset of the relation defined by S . This means that P never does anything that S couldn't do. However, P doesn't have to do everything that S can do. Code for square root is probably deterministic and always returns the same result for a given argument. Even though the spec allows several results (all the ones that are within the specified tolerance), we don't require code for to be able to produce all of them; instead we are satisfied with one.

Actually this is not enough. The definition we have given allows P 's relation to be empty, that is, it allows P not to terminate. This is usually called 'partial correctness'. In addition, we usually want to require that P 's relation be total on the domain of S ; that is, P must produce some result whenever S does. The combination of partial correctness and termination is usually called 'total correctness'.

If we are only interested in external behavior of a procedure that is part of a stateless module, the only state we care about is the arguments and results of the procedure. In this case, a transition is completely described by a single entry in a trace, such as $(\text{Read}(a1), v)$.

Now we are ready to consider modules with state. Our idea is to generalize what we did with pairs of states described by single trace entries to sequences of states described by longer traces. Suppose that T and S are any modules that have the same external interface (set of procedures that are exported and hence may be invoked externally). In this discussion, we will often refer to S as the *spec* module and T as the *code*. Then we say that T implements S if every trace of T is also a trace of S . That is, the set of traces generated by T is a subset of the set of traces generated by S .

This says that any external behavior of the *code* T must also be an external behavior of the spec S . Another way of looking at this is that we shouldn't be able to tell by looking at the *code* that we aren't looking at the spec, so we have to be able to explain every behavior of T as a possible behavior of S .

The reverse, however, is not true. We do not insist that the *code* must exhibit every behavior allowed by the spec. In the case of the simple memory the spec is completely deterministic, so the code cannot take advantage of this freedom. In general, however, the spec may allow lots of behaviors and the *code* choose just one. The spec for sorting, for instance, allows any sorted sequence as the result of `Sort`; there may be many such sequences if the ordering relation is not total. The *code* will usually be deterministic and return exactly one of them, so it doesn't exhibit all the behavior allowed by the spec.

Safety and liveness

Just as with procedures, this subset requirement is not strong enough to satisfy our intuitive notion of *code*. In particular, it allows the set of traces generated by T to be empty; in other words, the *code* might do nothing at all, or it might do some things and then stop. As we saw, the analog of this for a simple sequential procedure is non-termination. Usually we want to say that the *code* of a procedure should terminate, and similarly we want to say that the *code* of a module should keep doing things. More generally, when we have concurrency we usually want the *code* to be *fair*, that is, to eventually service all its clients, and more generally to eventually make any transition that continues to be enabled.

It turns out that any external behavior (that is, any set of traces) can be described as the intersection of two sets of traces, one defined by a *safety* property and the other defined by a *liveness* property.¹ A safety property says that in the trace nothing bad ever happens, or more precisely, that no bad transition occurs in the trace. It is analogous to partial correctness for a stateless procedure; a state machine that never makes a bad transition can define any safety property. If a trace doesn't satisfy a safety property, you can always find this out by looking at a finite prefix of the trace, in particular, at a prefix that includes the first bad transition.

A liveness property says that in the trace something good *eventually* happens. It is analogous to termination for a stateless procedure. You can never tell that a trace doesn't have a liveness property by looking at a finite prefix, since the good thing might happen later. A liveness property cannot be defined by a state machine. It is usual to express liveness properties in terms of *fairness*, that is, in terms of a requirement that if some transition stays enabled continuously it eventually occurs (weak fairness), or that if some transition stays enabled intermittently it eventually occurs (strong fairness).

With a few exceptions, we don't deal with liveness in this course. There are two reasons for this. First, it is usually not what you want. Instead, you want a result within some time bound, which is a safety property, or you want a result with some probability, which is altogether outside the framework we have set up. Second, liveness proofs are usually hard.

Abstraction functions and simulation

The definition of 'implements' as inclusion of external behavior is a sound formalization of our intuition. It is difficult to work with directly, however, since it requires reasoning about infinite sets of infinite sequences of actions. We would like to have a way of proving that T implements S that allows us to deal with one of T 's actions at a time. Our method is based on *abstraction functions*.

An abstraction function maps each state of the *code* T to a state of the *spec* S . For example, each state of the `WBCache` module gets mapped to a state of the `Memory` module. The abstraction function explains how to interpret each state of the *code* as a state of the *spec*. For example, Figure 3 depicts part of the abstraction function from `WBCache` to `Memory`. Here is its definition in `Spec`, copied from handout 5.

```
FUNC AF() -> M = RET (\ a | c!a => c(a) [*] m(a) )
```

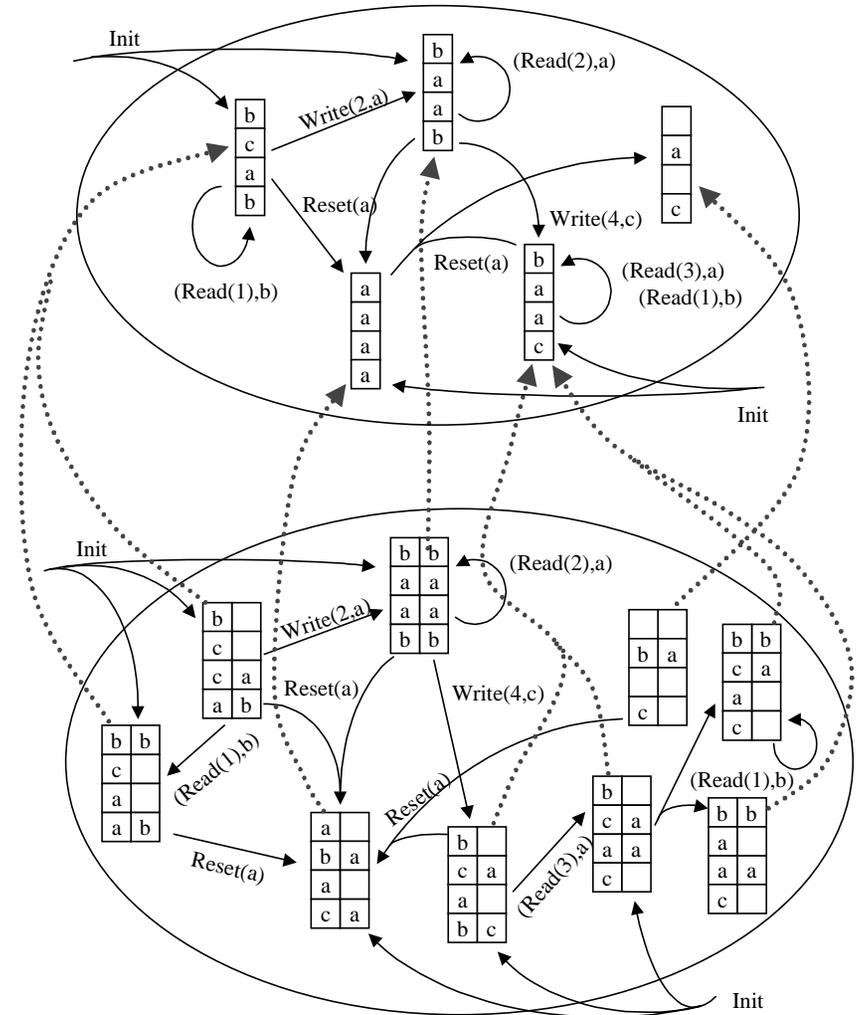


Figure 3: Abstraction function for `WBCache`

¹ B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing* 2, 3 (1987), pp 117-126.

You might think that an abstraction function should map the other way, from states of the *spec* to states of the *code*, explaining how to represent each state of the spec. This doesn't work, however, because there is usually more than one way of representing each state of the spec. For example, in the `wbCache` code for `Memory`, if an address is in the cache, then the value stored for that address in memory does not matter. There are also choices about which addresses appear in the cache. Thus, many states of the code can represent the same state of the spec. In other words, the abstraction function is many-to-one.

An abstraction function F is required to satisfy the following two conditions.

1. If t is any initial state of T , then $F(t)$ is an initial state of S .
2. If t is a reachable state of T and (t, π, t') is a step of T , then there is a step of S from $F(t)$ to $F(t')$, having the same trace.

Condition 2 says that T *simulates* S ; every step of T faithfully copies a step of S . It is stated in a particularly simple way, forcing the given step of T to simulate a single step of S . That is enough for the special case we are considering right now. Later, when we consider concurrent invocations for modules, we will generalize condition 2 to allow any number of steps of S rather than just a single step.

The diagram in Figure 4 represents condition 2. The dashed arrows represent the abstraction function F , and the solid arrows represent the transitions; if the lower (double) solid arrow exists in the code, the upper (single) solid arrow must exist in the spec. The diagram is sometimes called a “commutative diagram” because if you start at the lower left and follow arrows, you will end up in the same state regardless of which way you go.

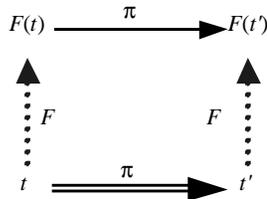


Figure 4: Commutative diagram for correctness

An abstraction function is important because it can be used to show that one module implements another:

Theorem 1: If there is an abstraction function from T to S , then T implements S , i.e., every trace of T is a trace of S .

Note that this theorem applies to both finite and infinite traces.

Proof: (Sketch) Let β be any trace of T , and let α be any execution of T that generates trace β . Use Conditions 1 and 2 above to construct an execution α' of S with the same trace. That is, if t is

the initial state of α , then let $F(t)$ be the initial state of α' . For each step of α in turn, use Condition 2 to add a corresponding step to α' .

More formally, this proof is an induction on the length of the execution. Condition 1 gives the basis: any initial state of T maps to an initial state of S . Condition 2 gives the inductive step: if we have an execution of T of length n that simulates an execution of S , any next step by T simulates a next step by S , so any execution of T of length $n+1$ simulates an execution of S .

We would like to have an inverse of Theorem 1: if every trace of T is a trace of S , then there is an abstraction function that shows it. This is not true for the simple abstraction functions and simulations we have defined here. Later on, in handout 8, we will generalize them to a simulation method that is strong enough to prove that T implements S whenever that is true.

Invariants

An *invariant* of a module is any property that is true of all *reachable* states of the module, i.e., all states that can be reached in executions of the module (starting from initial states). Invariants are important because condition 2 for an abstraction function requires us to show that the code simulates the spec from every reachable state, and the invariants characterize the reachable states. It usually isn't true that the code simulates the spec from every state.

Here are examples of invariants for the `HashMemory` and `MajorityRegister` modules, written in `Spec` and copied from handout 5.

```

FUNC HashMemory.Inv(nb: Int, m: HashT, default: V) -> Bool = RET
  ( nb > 0
    /\ m.size = nb
    /\ (ALL a | a.hf IN m.dom)
    /\ (ALL i :IN m.dom, p :IN m(i).rng | p.a.hf = i)
    /\ (ALL a | { j :IN m(a.hf) | m(a.hf)(j).a = a }.size <= 1) )

FUNC MajorityRegister.Inv(m: M) -> Bool = RET
  (ALL p :IN m.rng, p' :IN m.rng | p.seqno = p'.seqno ==> p.v = p'.v)
  /\ (EXISTS maj | (ALL i :IN maj, p :IN m.rng | m(i).seqno >= p.seqno))

```

For example, for the `HashMemory` module, the invariant says (among other things) that a pair containing address `a` appears only in the appropriate bucket `a.hf`, and that at most one pair for an address appears in the bucket for that address.

The usual way to prove that a property P is an invariant is by induction on the length of finite executions leading to the states in question. That is, we must show the following:

(Basis, length = 0) P is true in every initial state.

(Inductive step) If (t, π, t') is a transition and P is true in t , then P is also true in t' .

Not all invariants are proved directly by induction, however. It is often better to prove invariants in groups, starting with the simplest invariants. Then the proofs of the invariants in the later groups can assume the invariants in the earlier groups.

Example: We sketch a proof that the property `MajorityRegister.Inv` is in fact an invariant.

Basis: In any initial state, a single (arbitrarily chosen) default value d appears in all the copies, along with the `seqno` 0. This immediately implies both parts of the invariant.

Inductive step: Suppose that (t, π, t') is a transition and `Inv` is true in t . We consider cases based on π . If π is an invocation or response, or the body of a `Read` procedure, then the step does not affect the truth of `Inv`. So it remains to consider the case where π is a `Write`, say `Write(v)`.

In this case, the second part of the invariant for t (i.e., the fact that the highest `seqno` appears in more than half the copies), together with the fact that the `Write` reads a majority of the copies, imply that the `Write` obtains the highest `seqno`, say i . Then the new `seqno` that the `Write` chooses must be the new highest `seqno`. Since the `Write` writes $i+1$ to a majority of the copies, it ensures the second part of the invariant. Also, since it associates the same v with the `seqno` $i+1$ everywhere it writes, it preserves the first part of the invariant.

Proofs using abstraction functions

Example: We sketch a proof that the function `WBCache.AF` given above is an abstraction function from `WBCache` to `Memory`. In this proof, we get by without any invariants.

For Condition 1, suppose that t is any initial state of `WBCache`. Then `AF(t)` is some (memory) state of `Memory`. But all memory states are allowable in initial states of `Memory`. Thus, `AF(t)` is an initial state of `Memory`, as needed. For Condition 2, suppose that t and `AF(t)` are states of `WBCache` and `Memory`, respectively, and suppose that (t, π, t') is a step of `WBCache`. We consider cases, based on π .

For example, suppose π is `Read(a)`. Then the step of `WBCache` may change the cache and memory by writing a value back to memory. However, these changes don't change the corresponding abstract memory. Therefore, the memory correspondence given by `AF` holds after the step. It remains to show that both `Reads` give the same result. This follows because:

The `Read(a)` in `WBCache` returns the value `t.c(a)` if it is defined, otherwise `t.m(a)`.

The `Read(a)` in `Memory` returns the value of `AF(t).m(a)`.

The value of `AF(t).m(a)` is equal to `t.c(a)` if it is defined, otherwise `t.m(a)`. This is by the definition of `AF`.

For another example, suppose π is `Write(a, v)`. Then the step of `WBCache` writes value v to location a in the cache. It may also write some other value back to memory. Since writing a value back does not change the corresponding abstract state, the only change to the abstract state is that the value in location a is changed to v . On the other hand, the effect of `Write(a, v)` in `Memory` is to change the value in location a to v . It follows that the memory correspondence, given by `AF`, holds after the step.

We leave the other cases, for the other types of operations, to the reader. It follows that `AF` is an abstraction function from `WBCache` to `Memory`. Then Theorem 1 implies that `WBCache` implements `Memory`, in the sense of trace set inclusion.

Example: Here is a similar analysis for `MajorityRegister`, using `MajorityRegister.AF` as the abstraction function.

```
FUNC AF() -> V = RET m.rng.fmax((\ p1, p2 | p1.seqno <= p2.seqno)).v
```

This time we depend on the invariant `MajorityRegister.Inv`. Suppose π is `Read(a)`. No state changes occur in either module, so the only thing to show is that the return values are the same in both modules. In `MajorityRegister`, the `Read` collects a majority of values and returns a value associated with the highest `seqno` from among that majority. By the invariant that says that the highest `seqno` appears in a majority of the copies, it must be that the `Read` in fact obtains the highest `seqno` that is present in the system. That is, the `Read` in `MajorityRegister` returns a value associated with the highest `seqno` that appears in state t .

On the other hand, the `Read` in `Register` just returns the value of the single variable x in state s . Since `AF(t) = s`, it must be that `s.x` is a value associated with the highest `seqno` in t . But the uniqueness invariant says that there is only one such value, so this is the same as the value returned by the `Read` in `MajorityRegister`.

Now suppose π is `Write(v)`. Then the key thing to show is that `AF(t') = s'`. The majority invariant implies that the `Write` in `MajorityRegister` sees the highest `seqno` i and thus $i+1$ is the new highest `seqno`. It writes $(i+1, v)$ to a majority of the copies. On the other hand, the `Write` in `Register` just sets x to v . But clearly v is a value associated with the largest `seqno` after the step, so `AF(t') = s'` as required.

It follows that `AF` is an abstraction function from `MajorityRegister` to `Register`. Then Theorem 1 implies that `MajorityRegister` implements `Register`.