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.

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, stripping, 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
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 proc-
essing, 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.

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.

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 com-
puting. The relevant ones for systems are SigArch for computer architecture, SigPlan for pro-
gramming 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 Si-
 Ops 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 Operating Systems 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; pub-
lished as special issues of ACM SigOps Operating Systems Review, SigArch Computer Ar-
chitecture 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 Com-
munication 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]
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 \( \text{Bool} \), which contains two elements true and false. Expressions in these operators (and the other ones introduced later) are called ‘propositions’.

**Basic operators.** These are \( \land \) (and), \( \lor \) (or), and \( \neg \) (not). The meaning of these operators can be conveniently given by a ‘truth table’ which lists the value of \( a \) op \( b \) for each possible combination of values of \( a \) and \( b \) (the operators on the right are discussed later) along with some popular names for certain expressions and their operands.

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( \neg a )</th>
<th>( a \land b )</th>
<th>( a \lor b )</th>
<th>( a \equiv b )</th>
<th>( a \implies b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name of ( a )</th>
<th>conjunct</th>
<th>disjunct</th>
<th>antecedent</th>
</tr>
</thead>
<tbody>
<tr>
<td>name of ( b )</td>
<td>conjunct</td>
<td>disjunct</td>
<td>consequent</td>
</tr>
</tbody>
</table>

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

Since the table has only four rows, there are only 16 Boolean operators, one for each possible arrangement of \( \land \) and \( \lor \) 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 \( \land \) and \( \lor \) operators are

- commutative and
- associative and
- distribute over each other.

That is, they are just like \( \ast \) (times) and \( + \) (plus) on integers, except that \( + \) doesn’t distribute over \( \ast \):

\[
    a + (b \ast c) \neq (a + b) \ast (a + c)
\]

but \( \lor \) does distribute over \( \land \):

\[
    a \lor (b \land c) = (a \lor b) \land (a \lor c)
\]

An operator that distributes over \( \land \) is called ‘conjunctive’; one that distributes over \( \lor \) is called ‘disjunctive’. Both \( \land \) and \( \lor \) are both conjunctive and disjunctive. This takes some getting used to.

The relation between these operators and \( \neg \) is given by DeMorgan’s laws (sometimes called the “bubble rule” by logic designers), which say that you can push \( \neg \) inside \( \land \) or \( \lor \) (or pull it out) by flipping from one to the other:

\[
    \neg (a \land b) = \neg a \lor \neg b
\]

\[
    \neg (a \lor b) = \neg a \land \neg b
\]

\(^1\) It’s possible to write all three in terms of the single operator ‘nor’ or ‘nand’, but our goal is clarity, not minimality.
To put a complex expression into a "disjunctive normal form" replace terms in \( \equiv \) and \( \Rightarrow \) with their equivalents in \( \land, \lor, \) and \( (\text{given below}), \) use DeMorgan's laws to push all the \( \sim \)'s in past \( \land \) and \( \lor \) so that they apply to variables, and then distribute \( \land \) over \( \lor \) so that the result looks like
\[
(a \land \sim b \land \ldots) \lor (\sim b \land a \land \ldots) \lor \ldots
\]
The disjunctive normal form is unique (up to ordering, since \( \land \) and \( \lor \) are commutative). Of course, you can also distribute \( \lor \) over \( \land \) to get a unique "conjunctive normal form".

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

Because \( \text{Bool} \) is the result type of relations like \( = \), you can write expressions that mix up relations with other operators in ways that are impossible for any other type. Notably
\[
(a \equiv b) = (((a \land b) \lor \sim(a \land b))
\]
Some people feel that the outer \( = \) in this expression is somehow different from the inner one, and write \( \equiv \). Experience suggests, however, that this is often a harmful distinction to make.

**Implication.** We can define an ordering on \( \text{Bool} \) with \( \text{false} > \text{true} \), that is, \( \text{false} \) is greater than \( \text{true} \). 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 "\( a \) 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 \( \Rightarrow \) to \( \lor \) by the rule
\[
(a \Rightarrow b) = \sim a \lor b
\]
and then just use the simpler rules for \( \land, \lor, \) 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 \( \text{false} \)).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.

2It sometimes seems odd that \( \text{false} \) implies \( b \) regardless of what \( b \) is, but the "\( \text{if} \ldots \text{then} \)" form makes it clearer what is going on: if \( \text{false} \) is \( \text{true} \) you can conclude anything, but of course it isn’t. A proposition that implies \( \text{false} \) 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.

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


Of course, implication has the properties we expect of an ordering:
- Transitive: If \( a \Rightarrow b \) and \( b \Rightarrow c \) then \( a \Rightarrow c \).4
- 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, or vice versa. Thus
\[
(a \Rightarrow b \lor c) = (a \land \sim b \Rightarrow c)
\]
As special cases in addition to the contrapositive we have
\[
(a \Rightarrow b) = (a \land \sim b \Rightarrow \text{false} ) = (\sim (a \land \sim b) \lor \text{false} = \sim a \lor b
\]
\[
(a \Rightarrow b) = (a \land (b \lor c) \Rightarrow \text{true} ) = (\sim a \lor b \Rightarrow \text{false}) \lor \sim a \lor b = \sim a \lor b
\]
since \( \text{false} \) and \( \text{true} \) are the identities for \( \lor \) and \( \land \).

We say that an operator \( \text{op} \) is 'monotonic' in an operand if replacing that operand with a stronger (or weaker) one makes the result stronger (or weaker). Precisely, "\( \text{op} \) is monotonic in its first operand" means that if \( a \Rightarrow b \) then \( (a \text{ op } c) \Rightarrow (b \text{ op } c) \). Both \( \land \) and \( \lor \) are monotonic; in fact, any operator that is conjunctive (distributes over \( \land \)) is monotonic, because if \( a \Rightarrow b \) then \( a \Rightarrow (a \land b) \), so
\[
(a \Rightarrow b \text{ op } c) = (a \land b \text{ op } c) \Rightarrow (a \text{ op } c \land b \text{ op } c) = (a \text{ op } c) \Rightarrow (b \text{ op } c)
\]
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"):
\[
\begin{align*}
top & = \text{false} \\
bottom & = \text{true} \\
meet & = \land \quad \text{least upper bound, so} \quad (a \land b) \Rightarrow a \quad \text{and} \quad (a \land b) \Rightarrow b \\
join & = \lor \quad \text{greatest lower bound, so} \quad a \Rightarrow (a \lor b) \quad \text{and} \quad b \Rightarrow (a \lor b)
\end{align*}
\]
This suggests two more expressions that are equivalent to \( a \Rightarrow b \):
\[
(a \Rightarrow b) = (a = (a \land b)) \quad \text{and} \quad \text{a} \Rightarrow \text{b} \Rightarrow \text{true}
\]
\[
(a \Rightarrow b) = (\sim b \Rightarrow \sim a) \quad \text{or} \quad \text{a} \Rightarrow \text{b} \Rightarrow \text{false}
\]
\[
\text{Predicate logic}
\]
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 \( 5 < 3 \Rightarrow \text{false} \). 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 notation:6
\[
\forall x \mid P(x) = P(x_1) \land P(x_2) \land \ldots \quad \text{for all } x, P(x).
\]
In Spec,
\[
\{ \forall x \mid P(x) \} \lor \{ \exists x \mid P(x) \}
\]

Handout 2. Overview and Background

7

8

Handout 2. Overview and Background
\[ \exists x \mid P(x) = P(x_1) \lor P(x_2) \lor \ldots \]

there exists an \( x \) such that \( P(x) \). In Spec,
\[
\{ \exists x \mid P(x) \} \lor \{ x \mid P(x) \}
\]

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:
\[
\sim \left( \forall x \mid P(x) \right) = \exists x \mid \sim P(x)
\]
\[
\sim \left( \exists x \mid P(x) \right) = \forall x \mid \sim P(x)
\]

Also, because \( \land \) and \( \lor \) are conjunctive and therefore monotonic, \( \lor \) and \( \exists \) are conjunctive and therefore monotonic.

It is not true that you can reverse the order of \( \lor \) 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 \( P.\text{func} \) (or \( P.\text{setF} \) for the set). \( P.\text{func} \) 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.

Summary of logic

The \( \land \) and \( \lor \) operators are commutative and associative and distribute over each other.

DeMorgan’s laws:
\[ \sim (a \land b) = \sim a \lor \sim b \]
\[ \sim (a \lor b) = \sim a \land \sim b \]

Any expression has a unique (up to ordering) disjunctive normal form in which \( \lor \) combines terms in which \( \lor \) combines (possibly negated) variables: \( (a_1 \land \sim a_2 \land \ldots) \lor (\sim b_1 \land b_2 \land \ldots) \lor \ldots \)

Implication:
\[ (a \Rightarrow b) = \sim a \lor b \]

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
\[
\text{top} = \text{false} \quad \text{so} \quad \sim \text{false} \Rightarrow \text{true}
\]
\[
\text{bottom} = \text{true} \quad \text{least upper bound, so} \quad (a \land \text{false}) \Rightarrow a
\]
\[
\text{meet} = \land \quad \text{greatest lower bound, so} \quad a \Rightarrow (a \lor \text{false})
\]

For all \( x \), \( P(x) \):
\[ \forall x \mid P(x) = P(x_1) \land P(x_2) \land \ldots \]

There exists an \( x \) such that \( P(x) \):
\[ \exists x \mid P(x) = P(x_1) \lor P(x_2) \lor \ldots \]

\(^7\) 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.
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.


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.

In the language outline, the parts in small type describe less important features, and you can skip them on first reading.

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:

- **communicate** The spec should tell both the client and the implementer what each needs to know.
- **precisely** We should be able to prove theorems or compile machine instructions based on the spec.
- **essential** Unnecessary requirements in the spec may confuse the client or make it more expensive to implement the system.
- **behavior** 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. It doesn’t have to be C or Java code; we will give lots of examples of code in Spec which would still require a lot of work on the details of data structures, memory allocation, etc. to turn it into an executable program. You might wonder what good this kind of high-level code is. It expresses the difficult parts of the design clearly, without the straightforward details needed to actually make it run.

**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, usually at least one for every possible input).

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
1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
...
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:

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

Here the expression in \(\ldots\) is very close to the usual mathematical notation for defining a set. Read it as “The set of all \(q\) which are sequences of integers such that \(q\).size = 18 and ...”. Spec sequences are indexed from 0. The \((\text{ALL } \ldots\)) is a universally quantified predicate, and \(==>\) stands for implication, since Spec uses the more familiar \(\Rightarrow\) 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. 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.

```spec
VAR i, j |
<< i := 1; j := 1 >> ;
DO << j < 18 == BEGIN i := i [i := i+1 END; Output(i); j := j+1 >> OD
```

Here there is a good deal of new notation, in addition to the familiar semicolons, assignments, and plus signs.
The combination of \( \text{VAR} \) and \( \rightarrow \) is a very common Spec idiom; read it as “choose a \( y \) such that \( \text{Abs}(x - y^2) < \text{eps} \) and do \( \text{RET} \) \( y \)”. Why is this the meaning? The \( \text{VAR} \) makes a choice of any \( \text{Real} \) as the value of \( y \), but the entire transition on the second line cannot occur unless the guard \( \text{Abs}(x - y^2) < \text{eps} \) is true. Hence the \( \text{VAR} \) must choose a value that satisfies the guard.

What can we learn from this example? First, the result of \( \text{SquareRoot0}(x) \) is not completely determined by the value of \( x \); any result whose square is within \( \text{eps} \) of \( x \) is possible. This is why \( \text{SquareRoot0} \) 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, \( \text{SquareRoot0} \) is non-deterministic.

Why did we write it that way? First of all, there might not be any \( \text{Real} \) (that is, any floating-point number of the kind used to represent \( \text{Real} \) whose square exactly equals \( x \)). We could accommodate this fact of life by specifying the closest floating-point number.\(^1\) Second, however, 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_):

\[
\text{APROC} \text{SquareRoot1}(x: \text{Real}) \rightarrow \text{Real} =
\begin{align*}
\langle \langle \text{VAR} y : \text{Real} | y >= 0 \rangle / \rangle & \text{Abs}(x - y^2) < \text{eps} \Rightarrow \text{RET} y \rangle \rangle
\end{align*}
\]

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

\[
\text{APROC} \text{SquareRoot2}(x: \text{Real}) \rightarrow \text{Real} ^{\text{RAISE} \text{S} \text{E} \text{S} \ (\text{undefined})} =
\begin{align*}
\langle \langle \text{VAR} y : \text{Real} | y >= 0 \rangle / \rangle & \text{Abs}(x - y^2) < \text{eps} \Rightarrow \text{RET} y \rangle \rangle \end{align*}
\]

\[^{1}\] This would still be non-deterministic in the case that two such numbers are equally close, so if we wanted a deterministic spec we would have to give a rule for choosing one of them, for instance, the smaller.

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.**

\[^{2}\] \( r := \text{SquareRoot1}(x) \).choose (using the function) is almost the same as \( r := \text{SquareRoot1}(x) \) (using the procedure). The difference is that because \text{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 \text{SquareRoots1} is given the same argument. The procedure, on the other hand, is non-deterministic and can return different values on successive calls, so that spec is weaker. Which one is more appropriate?
An atomic command is still simple, because its meaning is just a relation between states. The relation may be partial: in some states there may be no way to execute the command. When this happens we say that the command is not enabled in those states. As we saw, the relation is not necessarily a function, since the command may be non-deterministic.

A non-atomic command is much more complicated than an atomic command, because:

- Taken in isolation, the meaning of a non-atomic command is a relation between an initial state and a history. A history is a whole sequence of states, much more complicated than a single final state. Again, many histories can stem from a single initial state.
- The meaning of the (parallel) 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 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 declared in a module has the name `m.i` throughout the program. A **class** is a module that can be instantiated many times to create many objects, much like a Java class.

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. They should be enough for the Spec you will read and write in this course, but they don’t answer every question about Spec; for those answers, read the reference manual and the handouts on Spec semantics.

Paragraphs in small print contain material that you might want to skip on first reading.

There is a one-page summary of the Spec language at the end of this handout.

**Expressions, types, and relations**

Expressions are for computing functions of the state.

A **Spec expression is**

- a constant
- a variable
- an invocation of a function on an argument

**and its value is**

- the constant
- the current value of the variable
- 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).

**Types**

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. If is a type, has its set of values.
- A set of functions called the **methods** of the type. There is convenient syntax for invoking methods on a value of the type. A method of type is lifted to a method of a set of 's, a function or a relation from to in the obvious way, by applying it to the set elements or the result of the function or relation, unless overridden by a different in the definition of the higher type. Thus if has a **square** method, .square = . We’ll see that this is a form of function composition.

Spec is strongly typed. This means that you are supposed to declare the types of your variables, just as you do in Java. In return the language defines a type for every expression\(^3\) 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 can be checked mechanically.

If is a type, you can omit it in a declaration of the identifiers , etc. Thus `VAR int1, bool2, char* | ...`

---

\(^3\) 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.
is short for

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

Note that this can be confusing in a declaration like \( t, u: \text{Int} \), where \( u \) has type \( \text{U} \), not type \( \text{Int} \).

If \( e \in \text{IN} \text{T.all} \) then \( e \text{ AS} \text{T} \) is an expression with the same value and type \( \text{T} \); otherwise it’s undefined. You can write \( e \text{ IS} \text{T} \) for \( e \in \text{IN} \text{T.all} \).

Spec has the usual types:

```
Int, Nat (non-negative Int), Bool
```

- functions \( \text{T} \rightarrow \text{U} \)
- relations \( \text{T} 
\rightarrow \rightarrow \text{U} \)
- records or structs \([f1: \text{T1}, f2: \text{T2}, ...]\)
- tuples \((\text{T1}, \text{T2}, ...)\)
- variable-length arrays called sequences, \( \text{SEQ} \text{T} \)

A sequence is actually a function whose domain is \([0, 1, \ldots, n-1]\) for some \( n \). A record is actually a function whose domain is the field names, as strings. In addition to the usual functions like "+", "/", Spec also has some less usual operations on these types, which are valuable when you want to suppress code details; they are called constructors and combinations and are described below.

You can make a type with fewer values using \text{SUCHTHAT}. For example,

```
\text{T = Int SUCHTHAT} 0 <= t \wedge t <= 4
```

has the value set \([0, 1, 2, 3, 4]\). Here the expression following \text{SUCHTHAT} is short for \( \{t \in \text{Int} \mid 0 <= t \wedge t <= 4\} \), a lambda expression (with \( \lambda \) for \( \lambda \)) that defines a function from \( \text{Int} \rightarrow \text{Bool} \), and has value type \( \text{T} \) if it’s an \( \text{Int} \) and the function maps it to \( \text{true} \). You can write this for the argument of \text{SUCHTHAT} if the type doesn’t have a name. The type \( \text{IN} s \), where \( s \) has type \( \text{SET T} \), is short for \( \text{SET T SUCHTHAT} \text{this IN s} \).

**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. For example, if \( s \) is a \( \text{SEQ T} \), \( s.\text{head} \) is \( \text{Sequence[T].Head}(s) \), which is just \( s\{}\{} \) (which is undefined if \( s \) is empty). You can see that it’s shorter to write \( s.\text{head} \).

```
\text{Note that when you write e.m the method m is determined by the static type of e, and not by the value as in most object-oriented languages.}
```

You can define your own methods by using \text{WITH}. For instance, by defining

```
\text{TYPE Complex = [re: Real, im: Real] SUCHTHAT} \end{verbatim}
```

- \text{Add: Add(c, c') and Mag: 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.}

\footnote{Of course, \( s\{}\{} \) is shorter still, but that’s an accident; there is no similar alternative for \( s.\text{tail} \).}
The **op** on the conditional Boolean operators means that, unlike all other operators, they do not evaluate their second argument if the first one determines the result. Thus \( \neg \neg (x) \) is false if \( \neg (x) \) is false, even if \( g(x) \) is undefined.

### Unary operators


<table>
<thead>
<tr>
<th>Op</th>
<th>Prec.</th>
<th>Argument/result types</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg )</td>
<td>6</td>
<td>( \text{Int} \to \text{Int} )</td>
<td>negation</td>
</tr>
<tr>
<td>( \sim )</td>
<td>3</td>
<td>( \text{Bool} \to \text{Bool} )</td>
<td>complement</td>
</tr>
<tr>
<td>( \text{PToR} )</td>
<td>5</td>
<td>( \text{T} \to \text{U} )</td>
<td>relation complement</td>
</tr>
</tbody>
</table>

### Relations

A relation \( r \) is a generalization of a function: an arbitrary set of ordered pairs, defined by a predicate, a total function from pairs to \( \text{Bool} \). Thus \( r \) can 
1. relate an element of its domain to any number of elements of its range (including none).
2. like a function, \( r \) has \( \text{dom}, \text{rng}, \text{and inv} \) methods (the inverse is obtained just by flipping the ordered pairs), and you can compose relations with \( \circ \).

#### Examples:

The relation \( < \) on \( \text{Int} \). Its domain and range are \( \text{Int} \), and its inverse is \( > \).

The relation \( r \) given by the set of ordered pairs \( s = \{("a", 1), ("b", 2), ("a", 3)\} \); \( r = s.\text{pred}.\text{PToR} \); that is, turn the set into a predicate on ordered pairs and the predicate into a relation. Its inverse \( r.\text{inv} = \{(1, "a"), (2, "b"), (3, "a")\} \), which is the sequence \( \{"a", "b", "a"\} \). Its domain \( r.\text{dom} = \{"a", "b"\} \); its range \( r.\text{rng} = \{1, 2, 3\} \).

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 \( \{x\} \to r \) for the set of values related to \( x \) by \( r \); see below).

A relation \( r \) has methods

- \( r.\text{setF} \) to turn it into a set function: \( r.\text{setF}(x) \) is the set of elements that \( r \) relates to \( x \). This is total.
- \( r.\text{fun} \) to turn it into a function: \( r.\text{fun}(x) \) is undefined unless \( r \) relates \( x \) to exactly one value. Thus \( r.\text{fun} = r.\text{setF}.\text{one} \).

If \( s \) is a set, \( s.\text{id} \) relates every member of the set to itself, and \( s.\text{rel} \) is a relation that relates true to each member of the set; thus it is \( s.\text{pred}.\text{inv}.\text{restrict}([\text{true}]) \). The relation’s \( \text{rng} \) method inverts this: \( s.\text{rel}.\text{rng} = s.\text{dom} \).
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).

A set `s` can be viewed as a total function `s.pred` on `T` that is `true` on the members of `s` (sometimes called the ‘characteristic function’), or as a relation `s. rel` from `true` to the members of the set, or as the identity relation `s.id` that relates each member to itself, or as the universal relation `s.univ` that relates all the members to each other.

You can compose a set `s` with a function or a relation to get another set, which is the image of `s` under the function or relation.
relation computes from a set: the extremal points where the relation takes the elements of the set; here you get them all, so there’s no need for an arbitrary choice. If you think of the graph induced by the closure of the relation, starting from the elements of the set, then the leaves are the nodes of the graph that have no outgoing edges (successors).

$s = \{3,1,5\}$, $s\text{.perms} = \{(3,1,5), (3,5,1), (1,5,3), (5,1,3), (1,3,5), (5,3,1)\}$, $s\text{.sort} = \{1,3,5\}$, $s\text{.max} = 5$, $s\text{.min} = 3$.

### Method call

<table>
<thead>
<tr>
<th>Result type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-&gt;Bool</td>
<td>${t</td>
</tr>
<tr>
<td>Bool</td>
<td>$s\text{.pred}$</td>
</tr>
<tr>
<td>T-&gt;T</td>
<td>${t1 \times t2</td>
</tr>
<tr>
<td>T-&gt;T</td>
<td>$s\text{.rel} * s\text{.rel}$</td>
</tr>
<tr>
<td>T-&gt;T</td>
<td>${st</td>
</tr>
<tr>
<td>Bool</td>
<td>$s\text{.pred}(t)$</td>
</tr>
<tr>
<td>Bool</td>
<td>$s1 \setminus s2 \equiv s1, \text{or equivalently } (\forall t</td>
</tr>
<tr>
<td>S</td>
<td>$s1 \setminus s2$</td>
</tr>
<tr>
<td>S</td>
<td>$s1 \setminus s2$</td>
</tr>
<tr>
<td>T-&gt;S</td>
<td>$s\text{.include}(T-&gt;U-&gt;V)$</td>
</tr>
<tr>
<td>T-&gt;S</td>
<td>$s\text{.univ}(T-&gt;U)$</td>
</tr>
<tr>
<td>S</td>
<td>$s\text{.choose}(T)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.pred}(T-&gt;U)$</td>
</tr>
<tr>
<td>S</td>
<td>$s\text{.perms}(SET Q)$</td>
</tr>
<tr>
<td>S</td>
<td>$s\text{.seq}(Q)$</td>
</tr>
<tr>
<td>Q</td>
<td>$s\text{.seq}(Q)$</td>
</tr>
<tr>
<td>Q</td>
<td>$s\text{.seq}(Q)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.pred}(T-&gt;U)$</td>
</tr>
<tr>
<td>S</td>
<td>$s\text{.perms}(SET Q)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.sort}(T)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.sort}(T)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.max}(T)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.min}(T)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.leaves}(r)$</td>
</tr>
<tr>
<td>T</td>
<td>$s\text{.combine}(f)$</td>
</tr>
</tbody>
</table>

### Functions

A function is a set of ordered pairs; the first element of each pair comes from the function’s 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. Thus a function is a relation in which each element of the domain is related to at most one thing. A function may be partial, that is, undefined at some elements of its domain. The expression $\lambda x . f(x)$ 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 in the section on constructors below. Another example is $f|\{0\}$, which is the same as $f$ except that it is 0 at $x$. If you have never seen a constructor 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

$$\lambda y . f(y)$$

Here ‘\’ is ‘lambda’, and the subexpression $(y = x) \Rightarrow 0 [*] f(y)$) is a conditional, modeled on the conditional commands we saw in the first section; its value is 0 if $y \neq x$ 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 $y \neq x$. 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, $g(t, u)$ doesn’t type-check, and $g(t, u)$ has type $V$. Obviously $f$ and $g$ are closely related, but they are not the same. Functions declared with more than one argument are a bit tricky; they are discussed in the section on tuples below.

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

```spec
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` subcommands 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 `\`:

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

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

```spec
(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 `.is` with the obvious meaning:

```spec
i .. j = (i, i+1, \ldots, j-1, j)
```

In the section on constructors below we see another way to write this:

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

as

```spec
(\ 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:

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

rather than:

```spec
(\ i:IN 0 .. 4 | factorial(i))
```

A function $f$ has methods $f\text{.dom}$ and $f\text{.rng}$ that yield its domain and range sets, $f\text{.inv}$ that yields its inverse (which is undefined at $y$ unless $f$ maps exactly one argument to $y$), and $f\text{.rel}$ that turns it into a relation (see below). $f\text{.restrict(s)}$ is the same as $f$ on elements of $s$ and undefined elsewhere. The `overlay` operator combines two functions, giving preference to the second:

```spec
(f1 + f2)(x) = f2(x) if that is defined and f1(x) otherwise. So f1(3 -> 24) = f + (3 -> 24).
```
If type $U$ has method $m$, then the function type $F = T \to U$ has a “lifted” method $m$ that composes $U.m$ with $f$, unless $f$ already has a $m$ method. $F.m$ is defined by

$$(f \mid (\ t \mid f(t).m))$$

so that $f.m = f \times U.m$. For example, $(1, 3, 5).square = (1, 9, 25)$. If $m$ takes a second argument of type $V$, then $F$ takes a second argument of the same type and uses it uniformly. This also works for sets and relations.

You can turn a relation into a function by discarding all the pairs whose first element is related to more than one thing:

$$(f \mid (\ t \mid (f!t \Rightarrow f (t) \times f'(t))))$$


takes a second argument of type $V$, then $F$ takes a second argument of the same type and uses it uniformly. A relation type $R = T \to> U$ is also lifted to $T \to SET U$. These are used to automatically supply the higher-order types with lifting methods.

A function type $F = T \to U$ also has a set of lifting methods that turn an $f$ into a function on $T \to SET U$, or $T \to> U$ by composition. This works for $F = (T, U) \to Bool$ as well; the lifted method also takes a $w$ and uses it uniformly. A relation type $R = T \to> U$ is also lifted to $T \to SET U$. These are used to automatically supply the higher-order types with lifting methods.

You can go back and forth between a relation $T \to> U$ and a function $T \to SET U$ with the $setF$ and $setRel$ methods:

$$r := s \to r \quad \text{with method } s$$

You can go back and forth between a relation $T \to> U$ and a function $T \to SET U$ with the $setF$ and $setRel$ methods:

You can go back and forth between a relation $T \to> U$ and a function $T \to SET U$ with the $setF$ and $setRel$ methods:

$$r := s \to r \quad \text{with method } s$$

As we have seen, there are several ways to view a set or a relation. Which one is best depends on what you want to do with it, and what is familiar and comfortable in your application. Often the choice of representation makes a big difference to the convenience and clarity of your code, just as the choice of coordinate system makes a big difference in a physics problem. The following tables summarize the different representations, the methods they have, and the conversions among them. The players are sets, functions, predicates, and relations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Converts to by</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f.liftSet$</td>
<td>$T \to U \quad S = SET T \to SET U$</td>
</tr>
<tr>
<td>$f.liftRel$</td>
<td>$T \to U \quad R = (T, U) \to Bool$</td>
</tr>
<tr>
<td>$f.liftFun$</td>
<td>$T \to U \quad F = T \to V \to (V \to T) \to (V \to U)$</td>
</tr>
<tr>
<td>$f.liftSetRel$</td>
<td>$T \to&gt; U \quad R = T \to&gt; V \to&gt; (V \to&gt; T) \to&gt; (V \to&gt; U)$</td>
</tr>
<tr>
<td>$f.setRel$</td>
<td>$T \to&gt; U \quad R = T \to&gt; V \to&gt; (V \to&gt; T) \to&gt; (V \to&gt; U)$</td>
</tr>
</tbody>
</table>

You can go back and forth between a relation $T \to> U$ and a function $T \to SET U$ with the $setF$ and $setRel$ methods:

$$r := s \to r \quad \text{with method } s$$

You can go back and forth between a relation $T \to> U$ and a function $T \to SET U$ with the $setF$ and $setRel$ methods:

$$r := s \to r \quad \text{with method } s$$

Handout 3.  Introduction to Spec  17  Handout 3.  Introduction to Spec  18
<table>
<thead>
<tr>
<th>Type</th>
<th>based on</th>
<th>equivalent to convert with</th>
<th>converts to with</th>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET T</td>
<td>T-&gt;Bool</td>
<td>.pred</td>
<td>.rel</td>
<td>Bool-&gt;T</td>
</tr>
<tr>
<td>T-&gt;Bool</td>
<td>SET T</td>
<td>.set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bool-&gt;T</td>
<td>SET T</td>
<td>.set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-&gt;&gt;U</td>
<td>(T,U)-&gt;Bool</td>
<td>.pred</td>
<td>.pred.set</td>
<td>.fun</td>
</tr>
<tr>
<td>T-&gt;U</td>
<td>(T,U)-&gt;Bool</td>
<td>.pred</td>
<td>.fun</td>
<td></td>
</tr>
<tr>
<td>SEQ T</td>
<td>Int-&gt;T</td>
<td>.inv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here is another way to look at it. Each of the types that label rows and columns in the following tables is equivalent to the others, and the entries in the table tell how to convert from one form to another.

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>set</th>
<th>predicate</th>
<th>relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET T</td>
<td>T-&gt;&gt;U</td>
<td>Bool-&gt;T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-&gt;Bool</td>
<td></td>
<td></td>
<td>.set</td>
<td></td>
</tr>
<tr>
<td>Bool-&gt;T</td>
<td></td>
<td></td>
<td>.set</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ Q = \text{Int->}T \ \text{SUCHTHAT} \ q.\text{dom} = \{i: \text{Int} \mid 0 \leq i \land i < q.\text{dom}.\text{max}\} \]

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: t.Fill(n),
- testing for prefix or sub-sequence (not necessarily contiguous): q1<=q2, q1<<=q2,
- lexical comparison, permuting, and sorting,
- filtering, iterating over, and combining the elements,
- making a sequence into a relation that holds exactly between successive elements, treating a sequence as a multiset with operations to:
  - count the number of times an element appears: q.count(t),
  - test membership: t IN q,
  - take differences:
    - q1 – q2
    - q1 – {t}
    - q1 + q2
    - q1 + x.inv * q2
      \[ q1 = \{A,B,C\}; q2 = \{A,B,C\}; x = \{3,4\}; q1 + q2 = \{A,B,C,D,E\} \]

The value of \( i \ldots j \) is the sequence of integers from \( i \) to \( j \).

To apply a function \( \tau \) to each of the elements of \( q \), just use composition \( q * \tau \).

The \(*\) operator concatenates two sequences.

\[ q1 + q2 = q1 + x.\text{inv} * q2, \text{where} \ x = (q1.\text{size}..q1.\text{size}+q2.\text{size}-1) \]

\[ q1 = \{A,B,C\}; q2 = \{D,E\}; x = \{3,4\}; q1 + q2 = \{A,B,C,D,E\} \]

You can test for \( q1 \) being a prefix of \( q1 \) with \( q1 <= q2 \), and for it being an arbitrary subsequence, not necessarily contiguous, with \( q1 <=< q2 \).

\[ q1 <= q2 = (q1 <= q2.\text{restrict}(q1.\text{dom})) \]

\[ q1 = \{A,B\}; q2 = \{A,B,C\} \]

\[ q2 = q2.\text{restrict} = q1 \]

\[ q1 <=< q2 = (\text{EXISTS} \ s: \text{SET Int} \mid s <= q2.\text{dom} \land q1 = s.\text{sort} * q2) \]

\[ q1 = \{A,C\}; q2 = \{A,B,C\}; \text{choose} \ s = \{0,2\} = \{0,1,2\} \]

\[ s = s.\text{sort} * q2 = q1 \]

\[ 0 \rightarrow A \rightarrow 0 \rightarrow A \rightarrow A \]

\[ 1 \rightarrow B \rightarrow 1 \rightarrow B \rightarrow B \]

\[ 2 \rightarrow C \rightarrow 2 \rightarrow C \rightarrow C \]

\[ 0 \rightarrow 0 \rightarrow 0 \rightarrow A \rightarrow A \rightarrow A \]

\[ 1 \rightarrow 1 \rightarrow 1 \rightarrow B \rightarrow B \rightarrow B \]

\[ 2 \rightarrow 2 \rightarrow 2 \rightarrow C \rightarrow C \rightarrow C \]
You can take a subsequence of size $n$ starting at $i$ with \( q \text{.seg}(i, n) \) and a subsequence from $i_1$ to $i_2$ with \( q \text{.sub}(i_1, i_2) \).

\[
\text{q.seg}(i, n) = (i .. i+n-1) \ast q
\]

\[
\text{q.sub}(i_1, i_2) = (i_1 .. i_2) \ast q
\]

You can select the elements of $q$ that satisfy a predicate $f$ with \( q \text{.filter}(f) \).

\[
\text{q.filter}(f) = (q \ast f) \text{.set.sort} \ast q
\]

Sets, functions, and sequences are good when you have many values of the same type. When you have values of different types, you need a tuple or a record (they are the same, except that a record allows you to name the different values). In Spec a record is a function from the string names of its fields to the field values, and an \( r(f) \) is short for \( r \text{.f} := e \) and \( r := r \{f \rightarrow e\} \).

Since a pair of $\text{SEQ} \ T$ is a function $0..1 \rightarrow T$ and $\text{SEQ} (T, T)$ is a function $0..n \rightarrow 0..1 \rightarrow T$, \( \text{zip} \) just reverses the order of the arguments.

You can apply a combining function $f$ successively to the elements of $q$ with \( q \text{.iterate}(f) \). To get the result of combining all the elements of $q$ with $f$ use \( q \text{.combine}(f) = q \text{.iterate}(f) \text{.last} \). The syntax $+: q$ is short for \( q \text{.combine}(T.\ast+) \); it works for any binary operator that yields a $T$.

\[
\text{SEQ} \ T \text{ has the same perms, fsort, sort, fmax, fmin, max, and min constructors as SET } T.
\]

**Records and tuples**

Sets, functions, and sequences are good when you have many values of the same type. When you have values of different types, you need a tuple or a record (they are the same, except that a record allows you to name the different values). In Spec a record is a function from the string names of its fields to the field values, and an $n$-tuple is a function from $0..n-1$ to the field values. There is special syntax for declaring records and tuples, and for reading and writing record fields:

\[
[f : T, g : U] \text{ declares a record with fields } r \text{ and } g \text{ of types } T \text{ and } U.
\]

\[
(T, U) \text{ declares a tuple with fields of types } T \text{ and } U.
\]

$rf$ is short for $r(\text{"f"})$, and $rf := e$ is short for $r := r(\text{"f"} \rightarrow e)$.

There is also special syntax for constructing record and tuple values, illustrated in the following example. Given the type declaration

\[
\text{TYPE Entry = } \{ \text{salary: Int, birthdate: String} \}
\]

we can write a record value

\[
\text{Entry(salary := 23000, birthdate := "January 3, 1955"})
\]

which is short for the function constructor

\[
\text{Entry("salary" -> 23000, "birthdate" -> "January 3, 1955"})
\]

The constructor

\[
(23000, "January 3, 1955")
\]

yields a tuple of type $\langle \text{Int, String} \rangle$. It is short for

\[
(0 \rightarrow 23000, 1 \rightarrow "January 3, 1955")
\]
This doesn’t work for a singleton tuple, since (x) has the same value as x. However, the sequence constructor [x] will do for constructing a singleton tuple, since a singleton SEQ T is also a singleton tuple; in fact, this is the only way to write the type of a singleton tuple, since (T) is the same as T because parentheses are used for grouping in types just as they are in ordinary expressions.

The type of a record is String->Any SUCHTHAT ..., and the type of a tuple is Nat->Any SUCHTHAT ... Here the SUCHTHAT clauses are of the form this("f") IS T; they specify the types of the fields. In addition, a record type has a method called fields whose value is the sequence of field names (it’s the same for every record). Thus [F: T, g: U] is short for

\[
\text{String->Any SUCHTHAT } \begin{cases} 
    \text{this.dom} -> \{"f", g"\} & \text{IF } \text{this("f") IS T} \\
    \text{this("g") IS U} & \text{ELSE }
\end{cases}
\]

A tuple type works the same way; its fields is just 0...n-1 if the tuple has n fields. Thus (T, U) is short for

\[
\text{Int->Any SUCHTHAT } \begin{cases} 
    \text{this.dom} = 0..1 & \text{IF } \text{this(0) IS T} \\
    \text{this(1) IS U} & \text{ELSE }
\end{cases}
\]

Thus to convert a record r into a tuple, write r.fields * r, and to convert a tuple t into a record, write r.fields.inv * t.

There is no special syntax for tuple fields, since you can just write t(2) and t(2) := e to read and write the third field, for example (remember that fields are numbered from 0).

Functions declared with more than one argument are a bit tricky; they take a single argument that is a tuple. So f(x: Int) takes an Int, but f(x: Int, y: Int) takes a tuple of type (Int, Int). This convention keeps the tuples in the background as much as possible. The normal syntax for calling a function is f(x, y), which constructs the tuple (x, y) and passes it to f. However, f(x) is treated differently, since it passes x to f, rather than the singleton tuple (x). If you have a tuple t in hand, you can pass it to f by writing f$t without having to worry about the singleton case; if t takes only one argument, then t must be a singleton tuple and f$t will pass t(0) to f. Thus f$x is the same as f(x, y) and f$s(x) is the same as f(x).

A function declared with names for the arguments, such as

\[
\begin{align*}
    & (i: \text{Int}, s: \text{String} \mid i + \text{StringToInt}(x)) \\
    & \text{argNames}
\end{align*}
\]

which has the names, (Int, String)->Int. However, it also has a method argNames that returns the sequence of argument names, ["i", "s"] in the example, just like a record. This makes it possible to match up arguments by name, as in the following example.

A database is a set of records. A selection query q is a predicate that we want to apply to records. To do this we get from the field names, which are strings, to the argument for q? Assume that q has an argNames method. So if \( r \ IN \ a.q.argNames \) is the tuple that we want to feed to q$5(q.q.argNames * r) is the query, where $ is the operator that applies a function to a tuple of its arguments.

There is one problem if not all fields are defined in all records: when we try to use q.q.argNames * r, it will be undefined if r doesn’t have all the fields that q wants. We want to apply it only to the records in \( a \) that have all the necessary fields. That is the set

\[
\begin{cases} 
    \{r : \text{IN } a \mid q.q.argNames <= r.fields\} & \text{IF } q \text{ is true} \\
    \{r : \text{IN } a \mid q.q.argNames <= r.fields \land q5(q.q.argNames * r)\} & \text{ELSE }
\end{cases}
\]

To project the database, discarding all the fields except the ones in projection (a set of strings), write

\[
\{r :\text{IN } a \mid r\text{.restrict(projection)}\}
\]

Constructors

Functions, sets, and sequences make it easy to toss large values around, and constructors are special syntax to make it easier to define these values. For instance, you can describe a database as a function db from names to data records with two fields:

\[
\begin{align*}
    & \text{TYPE DB} = \{\text{String -> Entry} \mid \text{String->Any SUCHTHAT ...}\} \\
    & \text{VAR db := DB{}}
\end{align*}
\]

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

\[
\begin{align*}
    & \text{db := db{ }"Smith" -> Entry{salary := 23000, birthdate := 1955} } \\
    & \text{using another function constructor. The value of the constructor is that the same as db except at the argument "Smith", where it has the value Entry{...}, which is a record constructor. This assignment could also be written}
\end{align*}
\]

\[
\begin{align*}
    & \text{db("Smith") := Entry{salary := 23000, birthdate := 1955} } \\
    & \text{which changes the value of the db function at "Smith" without changing it anywhere else. This is actually a shorthand for the previous assignment. You can omit the field names if you like, so that}
\end{align*}
\]

\[
\begin{align*}
    & \text{db("Smith") := Entry{23000, 1955} } \\
    & \text{has the same meaning as the previous assignment. Obviously this shorthand is less readable and more error-prone, so use it with discretion. Another way to write this assignment is}
\end{align*}
\]

\[
\begin{align*}
    & \text{db("Smith").salary := 23000; db("Smith").birthdate := 1955}
\end{align*}
\]

A record is actually a function as well, from the strings that represent the field names to the field values. Thus Entry{salary := 23000, birthdate := 1955} is a function r: String->Any defined at two string values, "salary" and "birthdate": r("salary") = 23000 and r("birthdate") = 1955. We could have written it as a function constructor Entry{"salary" => 23000, "birthdate" => 1955}, and r.salary is just a convenient way of writing r("salary").

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

\[
\text{\{n : String \mid db)n,}
\]

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

\[
\text{\{n : String \mid \text{dbIn || n.size}\}}
\]
This three part set constructor contains i if and only if there exists an n such that \( db!n \) and 
\[ i = n.size. \]
So \( i :n.size \) is short for \( \{ n : \text{String} \mid db!n \} \). 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 \( \text{true} \) when there’s an edge from the first to the second, then
\[
\{ \text{n1: Node, n2: Node \mid } \text{graph(n1, n2)} \mid n2 \}.
\]
is the set of nodes that are the target of an edge, and the \( \mid n2 \) could be omitted. This is just the range \( \text{graph.rng} \) of the relation graph on nodes.

Following standard mathematical notation, you can also write
\[
\{ f : \text{IN openFiles} \mid f.\text{modified} \}
\]
to get the set of all open, modified files. This is equivalent to
\[
\{ f : f \text{ IN openfiles} \mid f.\text{modified} \}
\]
because if \( f \) is a SET \( T \), then \( \text{IN} f \) is a type whose values are the \( T \)’s in \( f \); in fact, it’s the type \( T \text{ SUCHTHAT} (\forall t : \text{IN} f) \}. This form also works for sequences, where the second operand of \( \text{IN} : \text{provides the ordering. So if } s \text{ is a sequence of integers, } (x : \text{IN} s \mid x > 0) \text{ is the positive ones, } (x : \text{IN} s \mid x > 0) \text{ is the squares of the positive ones, and } (x : \text{IN} s \mid x \times x) \text{ is the squares of all the integers, because an omitted predicate defaults to true.}\]  

To get sequences that are more complicated you can use sequence generators with \( \text{BY} \) and \( \text{WHILE} \). You can skip this paragraph until you need to do this.
\[
\begin{align*}
   i & := 1 \text{ BY } i + 1 \text{ WHILE } i < 5 \mid \text{true} \mid i \\
   (i, 1, 2, 3, 4, 5); \text{the second and third parts could be omitted. This is just like the “for” construction in C. An omitted } & \text{WHILE} \text{ defaults to } \text{true}, \text{ and an omitted } \text{ := defaults to an arbitrary choice for the initial value. If you write } \text{several generators, each variable gets a new value for each value produced, but the second and later variables are initialized first. So to get the sums of successive pairs of elements of } s, \text{ write} \}
\end{align*}
\]
\[
\begin{align*}
   s & := s \text{ BY } s.\text{tail} \text{ WHILE } s.\text{size} > 1 \mid (x(0) + x(1)) \\
   (1 \text{ BY } s \text{ IN s, sum := } 0 \text{ BY sum } + x) \\
   \text{Taking last of this would give the sum of the elements of } s. \text{ To get a sequence whose elements are reversed from those of } s, \text{ write} \}
\end{align*}
\]
\[
\begin{align*}
   s & := s \text{ BY } s.\text{tail} \text{ WHILE } s.\text{size} > 1 \mid (x(0) + x(1)) \\
   (1 \text{ BY } s \text{ IN s, rev := } () \text{ BY } (x) + rev.\text{last}) \\
   \text{To get the sequence } s, f(s), f^2(s), \ldots, f^n(s), \text{ write} \}
\end{align*}
\]
\[
\begin{align*}
   i & := 1 \text{ BY } n, \text{ iter := } e \text{ BY } f(\text{iter}) \\
   (i, n \text{ IN n, rev := } () \text{ BY } (x) + rev.\text{last}) \\
   \text{To get the sequence } (e, f(e), f^2(e), \ldots, f^n(e), \text{ write} \}
\end{align*}
\]

Combinations

A combination is a way to combine the elements of a non-empty sequence or set into a single value using an infix operator, which must be associative, and must be commutative if it is applied to a set. You write “operator : sequence or set”. This is short for \( q.\text{combine}(T.\text{operator}) \). Thus
\[
\begin{align*}
   + : \text{(SEQ String)}\{\text{"He", \text{"i", \text{"lo"}} = \text{"He" + \"l" + \"lo" = \"Hello} \text{ because } + \text{ on sequences is concatenation, and} \\
   \begin{align*}
   + : \{ i : 1 \ldots 4 \mid i**2 \} = 1 + 4 + 9 + 16 = 30 \\
   \text{Existential and universal quantifiers make it easy to describe properties without explaining how to test for them in a practical way. For instance, a predicate that is true iff the sequence } s \text{ is sorted is} \\
   (\text{ALL i : 1 .. } s.\text{size-1} \mid s(i-1) <= s(i)) \text{ This is a common idiom; read it as} \]
\]

3 In the sequence form, \( \text{IN} s \) is not a set type but a special construct; treating it as a set type would throw away the essential ordering information.
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

\[ p \implies c \]

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.

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{SKIP} & \text{SKIP} \\
\end{array}
\]

\[ y := 1 \]

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{SKIP} & \text{SKIP} \\
\end{array}
\]

\[ y = 0 \implies y := 1 \]

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{SKIP} & \text{SKIP} \\
\end{array}
\]

\[ y = 0 \implies y := 1 \]

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

\[ p \implies c \]

\[ p' \implies c' \]

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

\[ p \implies c \]

\[ \{ \} c' \]

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

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{SKIP} & \text{SKIP} \\
\end{array}
\]

\[ x = 0 \implies \text{SKIP} \]

\[ \{ \} y = 0 \implies y := 1 \]

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 \( \text{VAR} \).

This is sometimes called ‘angelic’ non-determinism, since the code finds all the possible transitions, yielding an outcome if any possible non-deterministic choice yield that outcome. This is usually what you want for a spec or high-level code; it is not so good for low-level code, since an operational implementation requires backtracking. The other kind of non-determinism is called ‘demonic’, it yields an outcome only if all possible non-deterministic choice yield that outcome. To do a command \( c \) and check that all outcomes satisfy some predicate \( p \), write

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[
\begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[ \{ \} \text{SKIP} \]

\[ \{ \} y = 0 \implies y := 1 \]

The precedence rules for commands are

- \( \text{EXCEPT} \) binds tightest
- \( ; \) next
- \( \implies \) next (for the right operand; the left side is an expression or delimited by \( \text{VAR} \))
- \( \{ \} \) bind least tightly.

These rules minimize the need for parentheses, which are written around commands in the ugly form \( \text{BEGIN} \ldots \text{END} \) or the slightly prettier form \( \text{IF} \ldots \text{FI} \); 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,

\[ p \implies \text{BEGIN} \ldots \text{END} \]

is the same as

\[ p \implies \text{BEGIN} \ldots \text{FI} \]

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

\[ \text{IF} \begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[ \{ \} \text{SKIP} \]

which means to do \( c1 \) if \( p \) is true, and then to do \( c2 \). The \( \{ \} \) \( \text{SKIP} \) ensures that the command before the \(~p\) does not fail, which would prevent \( c2 \) from getting done. Without the \( \{ \} \) \( \text{SKIP} \), that is in

\[ \begin{array}{c|c}
00 & 00 \\
01 & 01 \\
10 & 10 \\
11 & 11 \\
\end{array}
\]

\[ \{ \} \text{SKIP} \]

\[ \{ \} \text{SKIP} \]

\[ \{ \} \text{SKIP} \]
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 \( p \Rightarrow c1 \) FI; \( c2 \) has the same meaning as \( p \Rightarrow \text{BEGIN} \ c1; \ c2 \text{ END} \), 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 \( c1 \) \text{EXCEPT} \( ex \Rightarrow c2 \) 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

\( \text{VAR} \) gives you more dramatic non-determinism than \( [\ldots] \). The most common use is in the idiom

\[
\text{VAR} \ x: T \ | \ P(x) \Rightarrow c
\]

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

\[
\text{VAR} \ x: T \ | \ c
\]

then \( \text{VAR} \) acts like ordinary variable declaration, giving an arbitrary initial value to \( x \).

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

\[
\text{IF} \ \text{VAR} \ n: \text{Nat}, x: \text{Nat}, y: \text{Nat}, z: \text{Nat} \ | \\
(n > 2 \land x^n + y^n = z^n) \Rightarrow \text{out} := n
\]

\(* \) out := 0

which is read: choose integers \( n, x, y, z \) such that \( n > 2 \) and \( x^n + y^n = z^n \), and assign \( n \) to \( \text{out} \); if there are no such integers, assign \( 0 \) to \( \text{out} \).

\( \text{EXISTS} \ n: \text{Int}, x: \text{Int}, y: \text{Int}, z: \text{Int} \ | \\
(n > 2 \lor x^n + y^n = z^n)
\]

This command succeeds iff there were no way to set \( \text{out} \) to one of the \( n \)’s that exist. We could also write

\[
\text{VAR} \ s := \{ \ n: \text{Int}, x: \text{Int}, y: \text{Int}, z: \text{Int} \ | \\
(n > 2 \land x^n + y^n = z^n)
\]

\( \text{out} := n \)

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

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

The precedence of \( \text{EXISTS} \) is higher than \( [\ldots] \), which means that you can string together different \( \text{VAR} \) commands with \( [\ldots] \) or \( [\ldots] \), but if you want several alternatives within a \( \text{VAR} \) you have to use \( \text{BEGIN} \ldots \text{END} \) or \( \text{IF} \ldots \text{FI} \). Thus

---

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

Handout 3. Introduction to Spec
Atomicity

Each \(<\ldots>\) 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 \(<\ldots>\), 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

FUNCTION Fact(n: Int) -> Int =
  IF n <= 1 THEN RET 1
  ELSE 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 names 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. In a method you can refer to the current object instance by self. 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
PROC mean() -> Int = RET sum/n
PROC 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 \(0 \ldots n\), represented in the standard way by a relation \(g\): \(g(v1, v2)\) is true if \(v2\) is a successor of \(v1\), that is, if there is an edge from \(v1\) to \(v2\). We want a topological sort of the vertexes, that is, a sequence that is a permutation of \(0 \ldots n\) in which \(v2\) follows \(v1\) whenever \(v2\) is a successor of \(v1\) in the relation \(g\). Of course this possible only if the graph is acyclic.

```
MODULE TopologicalSort EXPORT V, G, Q, TopSort =

TYPE V = IN 0 .. n
  G = (V, V) -> Bool
  Q = SEQ V

FUNCTION TopSort(g) -> Q RAISES {cyclic} =
  IF VAR q | q IN (0 .. n).perm / IsTSorted(q, g) THEN RET q
  ELSE RAISE cyclic  FI

FUNCTION IsTSorted(q, g) -> Bool =
  IF 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 \texttt{perms} method on sets and sequences is defined in section 9 of the reference manual; the \texttt{TopSort} is a procedure, not a function, because its result is non-deterministic; we discussed this point earlier when studying \texttt{Square-Root}. Like that one, this spec has no internal state, since the module has no \texttt{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 \texttt{TopSort} is called.

\begin{verbatim}
MODULE TopSortImpl EXPORT V, G, Q, TopSort =
  \% implements TopSort
  \% plus the spec's types
  \% every vertex starts white
  \% every vertex starts white

  PROC TopSort(g) -> Q RAISES {cyclic} =
  \% visit every unvisited vertex
  \% visit every unvisited vertex
  \% grey — partly visited
  \% grey — partly visited
  \% add v to front of out
  \% add v to front of out

  PROC Visit(v, g) RAISES {cyclic} =
  \% add v to front of out
  \% add v to front of out

  PROC Locate(x) RAISES {cyclic} =
  \% locate x
  \% locate x

  PROC Split(x) RAISES {cyclic} =
  \% locate x
  \% locate x

  PROC Replace(from: X, size: X, b': B, from': X, size': X) =
  \% fails if it touches C's that aren't there.
  \% fails if it touches C's that aren't there.

  CLASS BufImpl EXPORT B, C, X, Get, Replace =
  \% implements Buffer

  TYPE N = X
  \% index in piece table

  MODULE BufImpl EXPORT B, C, X, Get, Replace =
  \% implements Buffer

  TYPE N = X
  \% index in piece table

  \% we could make the cost of Replace logarithmic as well, but to
  \% keep things simple we won't do that. See FSImpl in handout for more on this point.
\end{verbatim}
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|efg|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 + b2 = 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(i) \ \ n2 -> n1 / \ pt(n1).x <= x \ \ ( x < pt(n2).x \ \ x > pt.last.x )
% The loop terminates because n2 - n1 > 1 => n1 < n < n2, wn2 - 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) / \ x)

END BufImpl

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

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; that is, we are changing the definition of \( c \) to include the new properties. Get and Replace remain the same. In addition, we need a third exported method \( \text{Apply} \) that applies to each character in a subsequence of the buffer a map function \( f : C \rightarrow C \). 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) =
VAR nl := Split(from), n2 := Split(from + size) |
pt := pt.sub(0 \ , n1 - 1) + pt.sub(nl, n2 - 1) * { ( p | p.map := p.map * map) |
+ pt.sub(n2, pt.size - 1) }

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

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**

\[
\begin{align*}
\text{Point} & : \{ \text{r: Intensity}, \text{g: Intensity}, \text{b: Intensity}\} \\
\text{Color} & : \{ \text{r: Intensity}, \text{g: Intensity}, \text{b: Intensity}\} \\
\text{Window} & : \text{P} \rightarrow \text{C}
\end{align*}
\]

**FUNCTION AdjustX(dx: Int) -> (P -> P) = RET (\ p p{x + := dx})**

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.

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; that is, we are changing the definition of \( c \) to include the new properties. Get and Replace remain the same. In addition, we need a third exported method \( \text{Apply} \) that applies to each character in a subsequence of the buffer a map function \( f : C \rightarrow C \). 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) =
VAR nl := Split(from), n2 := Split(from + size) |
pt := pt.sub(0 \ , n1 - 1) + pt.sub(nl, n2 - 1) * { ( p | p.map := p.map * map) |
+ pt.sub(n2, pt.size - 1) }

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

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**

\[
\begin{align*}
\text{Point} & : \{ \text{r: Intensity}, \text{g: Intensity}, \text{b: Intensity}\} \\
\text{Color} & : \{ \text{r: Intensity}, \text{g: Intensity}, \text{b: Intensity}\} \\
\text{Window} & : \text{P} \rightarrow \text{C}
\end{align*}
\]

**FUNCTION AdjustX(dx: Int) -> (P -> P) = RET (\ p p{x + := dx})**

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.
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 on the screen, called V’s.

\[
\text{TYPE } WN = \text{IN } 0..n-1 \quad \% \text{ Window Number} \\
V = [w, \text{ offset: } P] \quad \% \text{ window on the screen} \\
\quad \text{ WITH } \{ c: (\{ v, p | v.w(p - v.offset) \}) \} \quad \% \text{C of a screen point p} \\
\text{VAR} \ ws: \text{SEQ V := \{} i: \text{IN } 0..n-1 \ | | \ V[i], P[10,5]) \} \quad \% \text{the Window System} \\
\text{FUNC Get(p) -> C = VAR wn := \{} w', c: (V.c(ws(wn'), p)).min | RET ws(wn).c(p) \\
\text{PROC Paint(wn, p, c) = ws(wn).w(p) := c} \\
\text{END Window}
\]

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, so in this code windows can’t move or get smaller. In a real window system an “expose” event tells a window to deliver the color of points that become newly visible.

The state is one \( v \) 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 \( P \rightarrow \text{SET WN} \). 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 \( ws \). Note that the abstract state \( ws \) is a sequence, that is, a function from window number to \( V \) for the window. The abstraction function gives the value of the \( wH \) function in terms of the code variables \( wH \) and \( offset \); that is, it is a function from \( wH \) and \( offset \) to \( ws \). By convention, we don’t write this as a function explicitly.

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 \( wH \) that is equal to \( ws(wH) \). This makes the abstraction function very simple, but then we need an invariant that says \( offset(wH) = wH(n).offset \). This is perfectly correct, but it’s usually better to put as little stuff in history variables as possible.

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

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 \(< i := i + 1 >>\) describes the transitions \( i=1 \rightarrow i=2, i=2 \rightarrow i=3, \text{etc.} \) (Actually, many transitions are summarized by \( i=1 \rightarrow i=2, \text{for instance, } (i=1, j=1) \rightarrow (i=2, j=1) \) 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 \(<< i := 1 [] i := 1 + 1 >>\) 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.


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 associated with additional information: Scope rules

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 $x\text{List}$ is defined by

$$x\text{List} ::= x \mid x\text{List}$$

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 (io) 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 union type and exactly one type in the union is a routine type. Note that this covers sequence, tuple, and record types. Under corresponding conditions we say that $e$ has a set type.

Two types are equal if their definitions are the same (that is, have the same parse trees) after all \texttt{WITH} clauses have been discarded. Type equality is intractable. 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 it’s OK to use a $T$ where a $U$ is required. This is true if the type-checker thinks they may have some non-trivial 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 $T'$ such that $F$ or $(\ldots + T' + \ldots)$ and $T'$ fits $U$, or vice versa. There may be no values in common, but the type-checker can't analyze the \texttt{SUCHTHAT} clauses to find out. There's a special case for the \texttt{SUCHTHAT} clauses of record and tuple types, which the type-checker can analyze: $T$’s \texttt{SUCHTHAT} must imply $U$’s.
- $T=T_1->T_2$ raises $EX_t$ and $U=U_1->U_2$ raises $EX_u$, or one or both raises are missing, and $U_1$ fits $T_1$ and $T_2$ fits $U_2$. Similar rules apply for \texttt{PROC} and \texttt{APROC} types. This also covers sequences. Note that the test is reversed for the argument types.
- $T=SET T'$ and $U=SET U'$ 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 “$U_1$ fits $T_1$” replaced by “$U_1$ includes $T_1$ and $EX_t$ is a superset of $EX_u$”. 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 \texttt{type-checking}. The rules for type-checking are given in sections 5 and 6.
The ambiguity of the type grammar is resolved by taking \( \rightarrow \) to be right associative and giving \( \text{WITH} \) and \( \text{RAISES} \) higher precedence than \( \rightarrow \).

1. A \( \text{SEQ} \ T \) is just a function from \( 0..\text{size}-1 \) to \( T \). That is, it is short for
   \[
   (\text{Int} \rightarrow \text{T}) \ \text{SUCHTHAT} \ (\ \forall \ f: \text{Int} \rightarrow \text{T} \ | \ \{ \exists \ \text{SIZE} \ \text{Int} \ | \ f.\text{do}m = 0..\text{size}-1\} \)
   \] 
   with \( \{ \text{see section 9} \} \).
   This means that \( \!, \) 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 \( \text{SEQ Char} \).

   % like an enumeration
   % sequence \( \text{char} \)

   % integers

   % naturals: non-negative integers

   % set

   % a tuple; \( \text{T} \) is the same as \( \text{T} \rightarrow \text{U} \).

   % restrict the value set \[ \text{IN exp} = \text{T} \ \text{SUCHTHAT} \ (\ \forall \ t: \text{T} \ | \ t \ \text{IN exp}) \]

   % type from a module \[ \text{id} [ \text{typeList} ] . \text{id} \]

   % the union of the types

   % a set of exceptions

   % a name uses only for procedures

   % the exceptions it can return

   % a set of exceptions

   % declared as an exception set

   % set difference

   % a method

   % the string must be an operator

   % name is a routine

   % if type \( v \) has method \( m \), then the function type \( V = \text{T} \rightarrow \text{U} \) has a \text{lifted} method \( m \) that composes \( U.m \) with \( v \), unless \( V \) already has a \( m \) method.

   % a method

   % a routine

   % a module

   % the first id denotes a module

   % the method of type

   % short for \( \text{id} \) declared globally

   % short for \( \text{id} \)

   % only for procedures

   % the exceptions it can return

   % a set of exceptions

   % declared as an exception set

   % set difference

   % a method

   % the string must be an operator

   % other than "+", 

   % a module

   % name is a routine

   % if type \( v \) has method \( m \), then the function type \( V = \text{T} \rightarrow \text{U} \) has a \text{lifted} method \( m \) that composes \( U.m \) with \( v \), unless \( V \) already has a \( m \) method. \( V, m \) is defined by

   \[
   \{ v \ \text{\&} \ (v \ \text{IN} v.\text{do}m \ \text{\&} \ vv.\text{do}m \ \text{\&} \ v(t).\text{m}(vv(t)))\}
   \]
Lifting also works for relations to \( U \), and therefore also for \( R = (T, U) \rightarrow \text{Bool} \) and

\[
\{(r, \bar{r}) | \langle t, x | x \in \{u | r(t, u) \land \bar{r}(t, w) \lor u.m(w)\}\}\}
\]

If \( U \) doesn’t have a method \( m \) but \( \text{Bool} \) does, then the lifting is done on the function that defines the relation, so that \( r \land r \land x \in \{u, w | r(t, u) \land \bar{r}(t, w) \lor u.m(w)\} \).

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 \( T \rightarrow \ldots \) and \( T \rightarrow \text{SUCHTHAT} f \).

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 \( \text{name} \) and its value in the current state, \( \text{state(name)} \). The form \( T.m \) (where \( T \) denotes a type) is also a name; it denotes the \( m \) method of \( T \). Note that if \( \text{name} \) is \( \text{id} \) and \( \text{id} \) is declared in the current module \( m \), then it is short for \( \text{m.id} \).
- invocation \( f(e) \): \( f \) must have a function (not procedure) type \( U \rightarrow T \) raises \( E \) 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 \( r \) 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 \( \text{SUCHTHAT} \); if the dynamic check fails the result is a fatal error). Then \( f(e) \) has type \( T \).

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

If both \( r \) and \( e \) are normal, the result of \( f(e) \) at \( re \) can be:

- A normal value, which becomes the result of \( f(e) \).
- An exception, which becomes the result of \( f(e) \). If \( r \) 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 \( \land, \lor /, \) and \( \rightarrow \) that are defined in terms of conditionals [13].
exp ::= primary
    prefixOp exp
    exp infixOp exp

prefixOp : exp
% exp’s elements combined by op

exp IS type
% (EXISTS x: type | exp = x)

exp AS type
% error unless (exp IS type)

primary ::= literal
    name
    primary . id
% method invocation or record field
    primary arguments
% function invocation
    constructor
% function or sequence constructor
    expList
% the arg is the tuple (expList)
    ( )
% tuple constructor
    name
% name denotes a func/seq/set type
    name { }
% name denotes a seq/record type
    primary { FieldDefList }
% record constructor
    primary { * -> result }
% function or sequence constructor
    ( LAMBDA signature = cmd )
% function with the local state
% short for (LAMBDA (d -> T=RET exp))
    ( \ declList | exp )
% set constructor
    { declList | pred || exp }
% set constructor
% set of all elements of pred
% set represented by seqGen
% sequence generator
    id := exp
% sequence generator

default

fieldDef ::= id := exp
% the function is undefined
result ::= empty
% the function yields exp
    RAISE exception
% the function yields exception

prefixOp ::= -
% (6)
exp
% (6)

argList ::= ( expList )
% the arg is the tuple
( )
% empty function

constructor ::= { }
% empty function/sequence/set
% sequence/set constructor
( expList )
% tuple constructor
{ }
% name denotes a func/seq/record type
{ }
% name denotes a seq/record type
{ FieldDefList }
% record constructor
{ * -> result }
% function or sequence constructor
( LAMBDA signature = cmd )
% function with the local state
% short for (LAMBDA (d -> T=RET exp))
( \ declList | exp )
% set constructor
{ declList | pred || exp }
% set constructor
% set of all elements of pred
% set represented by seqGen
% sequence generator
id := exp
% sequence generator
id := exp BY exp WHILE exp
% sequence generator
id := exp
% sequence generator

( precedence )

infixOp ::= **
% (8)

argument/result types

operation

 exponentiate
 multiply
 function composition
 divide
 remainder
 add
 concatenation
 function overlay
 subtract
 set difference;
 function difference;
 function is defined
 apply func to tuple
 subranges
 less than or equal
 greater than
 less than
 greater or equal
 equal
 not equal
 non-contiguous sub-seq
 membership
 conditional and
 union
 conditional or
 conditional implies
 not one of the above
 negation
 complement
 not one of the above
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 is T."prefixOp"(e), where T is e’s type, and of infixOps e1 op 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 1 + { i : Int | 0 < i && i < 5 || i**2 } = 1 + 4 + 9 + 16 = 30, and if s is a sequence of strings, + : s 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].


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, (ALL d | p) = /\ : (d | p), and (EXISTS d | p) = /\ : (d | 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 if 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 (SEQ T + SET T), and its value is the sequence containing the values of the exps 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(7->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 \ declList | exp is short for (LAMBDA (declList) -> T = RET 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 if (EXISTS declList | pred \ x = exp). Thus

\{ i: Int | 0 < i && i < 5 || 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: Int | 0 < i && 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 \ segGenList | pred || exp \ has type SEQ T, where exp has type T in the current state augmented by segGenList, as follows. The value of

\{ x1 := e01 BY e1 WHILE p1, ..., xn := e0n BY en WHILE pn | pred || 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 =>```
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 $\langle\ldots\rangle$ brackets.

The meaning of an atomic command is a set of possible transitions, roughly one for each assignment.

Examples

\[
\begin{align*}
&\{i := 0\ BY \ i+1\ WHILE\ i \leq n\} = 0..n = \{0, 1, ..., n\} \\
&\{x \assign s,\ \text{sum} := 0\ BY\ \text{sum} + x\} \text{ reverse of } s \\
&\{x \assign s\ |\ \text{rev} = [x] + \text{rev}\}.\text{last} \\
&\{x \assign s\ |\ \text{rev} = [x]\}.\text{last} \\
&\{x \assign [s]\} = s \\
&\{x \assign \text{head}\ BY\ x.\text{next}\ \text{WHILE}\ x \# \text{nil}\ |\ r.\text{val}\} = \text{val fields of a list starting at head}
\end{align*}
\]

These operations are defined in section 9.

[12] These operations are defined in terms of conditionals:

\[
\begin{align*}
e1 \ OR\ e2 &= (e1 \Rightarrow \text{true} \ [\ast] e2) \\
e1 \ AND\ e2 &= (e1 \Rightarrow \text{false} \ [\ast] e2) \\
e1 \ OR\ e2 &= (e1 \Rightarrow \text{true} \ [\ast] e2)
\end{align*}
\]

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 $\exp\ IS\ \text{type}$ the value of $\exp\ AS\ \text{type}$ is the value of $\exp$, but its type is $\text{type}$ rather than the type of $\exp$.

[15] $\text{fst}$ applies the function $f$ to the tuple $t$. It differs from $f(t)$, which makes a tuple out of the list of expressions in $t$ and applies $f$ to that tuple.
You can only assign to a name declared with \texttt{VAR} or in a \texttt{signature}. In an assignment the \texttt{exp} must fit the type of the \texttt{lhs}, or there is a fatal error. In a function body \texttt{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,
\begin{verbatim}
  lhs(arguments) := exp is short for for lhs := lhs[arguments->exp], and
  lhs . id := exp is short for for lhs := lhs[id := exp].
\end{verbatim}
These abbreviations are expanded repeatedly until \texttt{lhs} is a name.

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

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

A guarded command fails if the result of \texttt{pred} is undefined or \texttt{false}. It is equivalent to \texttt{cmd} if the result of \texttt{pred} is \texttt{true}. A \texttt{pred} is just a Boolean \texttt{exp}; see section 4.

\begin{verbatim}
  S1 [ ] S2 chooses one of the \texttt{S1} to execute. It chooses one that doesn't fail. Usually \texttt{S1} and \texttt{S2} will be guarded. For example,
  x=1 => y:=0 [ ] x> 1 => y:=1
  and has no outcome if \texttt{x<1}. \texttt{S1} [ ] \texttt{S2} is the same as \texttt{S1} unless \texttt{S1} fails, in which case it's the same as \texttt{S2}.
\end{verbatim}

\begin{verbatim}
If \texttt{... FI} are just command brackets, but it often makes the program clearer to put them around a sequence of guarded commands, thus:
\begin{verbatim}
  IF \texttt{x < 0} \texttt{=>} \texttt{y := 3} [ ] \texttt{x = 0} \texttt{=>} \texttt{y := 4} [ ] \texttt{y := 5} \texttt{FI}
\end{verbatim}

\end{verbatim}

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

In an atomic command the atomic brackets can be used for grouping instead of \texttt{BEGIN ... END}; since the command can't be any more atomic, they have no other meaning in this context.

Execute \texttt{cmd} repeatedly until it fails. If \texttt{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.

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

The empty command and \texttt{SKIP} make no change in the state. \texttt{HAVOC} produces an arbitrary outcome from any state; if you want to specify undefined behavior when a precondition is not satisfied, write \texttt{-precondition \texttt{HAVOC}}.

A \texttt{RET} may only appear in a routine body, and the \texttt{exp} must fit the result type of the routine. The \texttt{exp} is omitted if the \texttt{return} of the routine's \texttt{signature} is empty.

For \texttt{arguments} see section 5. The argument are passed by value, that is, assigned to the formal of the procedure \texttt{A} function body cannot invoke a \texttt{PROC} or \texttt{APROC} together with the rule for \texttt{assignments} (see [7]) this ensures that it can't affect the state. An atomic command can invoke an \texttt{APROC} but not a \texttt{PROC}. A command is atomic if it is \texttt{<< cmd >>}, a subcommand of an atomic command, or one of the simple commands \texttt{SKIP}, \texttt{HAVOC}, \texttt{RET}, or \texttt{RAISE}. The type-checking rule for \texttt{invocations} is the same as for function invocations in expressions.
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 \texttt{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 \texttt{ex} must be in the \texttt{RAISES} set of a routine \texttt{r} if either \texttt{RAISE ex} or an invocation of a routine with \texttt{ex} in its \texttt{RAISES} set occurs in the body of \texttt{r} outside the scope of a handler for \texttt{ex}.

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

\section{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 \texttt{mformals} after the module \texttt{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 \texttt{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 \texttt{Global} scope for the identifiers declared at the top level of the program. An identifier \texttt{id} declared at the top level of a non-parameterized module \texttt{m} is short for \texttt{m.id} when it occurs in \texttt{m}. If it appears in the \texttt{exports}, it can be denoted by \texttt{m.id} anywhere. When an identifier \texttt{id} that is declared globally occurs anywhere, it is short for \texttt{Global.id}. \texttt{Global} cannot be used as a module \texttt{id}.

An exported \texttt{id} must be declared in the module. If an exported \texttt{id} has a \texttt{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 \texttt{WITH} clause, all its methods and constructors are accessible. This is Spec’s version of data abstraction.

\begin{verbatim}
program ::= toplevel* module* END
module ::= modclass id mformals exports = body END id
modclass ::= MODULE CLASS % [4]
exports ::= EXPORT exportList
export ::= id
mformals ::= empty [ mfpList ]
mfp ::= id
id WITH {methodList} % see section 4 for method
body ::= toplevel* id [ typeList ]
toplevel ::= VAR declInit* 
CONST declInit* 
routineDecl
EXCEPTION exSetDecl* 
TYPE typeDecl* 

routineDecl ::= FUNC id signature = cmd 
APROC id signature =<<cmd>> % function
PROC id signature = cmd 
ThRED id signature = cmd 

exSetDecl ::= id = exceptionSet

signature ::= ( declList ) returns raises 
( ) returns raises

typeDecl ::= id = type 
id = ENUM [ idList ]

[1] The \texttt{=} in a \texttt{constInit} (defined in section 6) specifies an initial value for the variable. The \texttt{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 \texttt{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 \texttt{exp} may be a procedure invocation as well as an ordinary expression. It’s a fatal error if the \texttt{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 \texttt{id} in the declaration together with the argument values. So

\begin{verbatim}
VAR sum := 0, count := 0;
THREAD P(i: Int) = i IN 0 .. 9 >>
  VAR t | t := P(i); <<sum := sum + t>>; <<count := count + 1>>
\end{verbatim}
\end{verbatim}
adds up the values of \( F(0) \ldots F(9) \) in parallel. It creates a thread \( P(i) \) for every integer \( i \); the threads \( P(0), \ldots, P(9) \) for which the guard is true invoke \( F(0), \ldots, F(9) \) in parallel and total the results in \( \text{sum} \). When \( \text{count} = 10 \) 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.

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.

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 \( \text{objs}(\text{self}).v \) 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 \( \text{objs}(\text{obj}) \). The invocation \( \text{obj}.m \) of \( \text{obj}.m(x) \) is short for \( \text{ObjMod}.m(\text{obj}) \) or \( \text{ObjMod}.m(\text{obj}, x) \) by the usual rule for methods, and it thus invokes the method m in \( \text{obj} \)’s body each occurrence of a class variable refers to the corresponding field in \( \text{obj} \)’s state.

\( \text{Obj}.\text{new()} \) returns a new and initialized Obj object. The following example shows how a class is transformed into a module.

6.826—Principles of Computer Systems  

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, ...]

Func f(p1: RT1, ...) = ... v1 ... Func f(self: Obj, p1: RT1, ...) = ... v1 ... 

Proc p(p2: RT2, ...) = ... v2 ... Proc p(self: Obj, p2: RT2, ...) = ... v2 ... 

Func AddT(t1, t2) = ... Func AddT(t1, t2) = ...

8. Scope

The declaration of an identifier is known throughout the smallest scope in which the declaration appears (redescription 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 toplevel (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 toplevel, the := of a record constructor, or the := or Br 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 toplevel in the current module m, it is short for m.id. If it refers to an identifier declared by a toplevel 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 toplevel), 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 (ALL any | s(any) =>>( any IS T))
% Defined everywhere so that type inclusion will work; see section 4.
WITH ("\"/":=Union, "/\":=Intersection, "-":=Difference, "+":=In, "":=Subset, choose:=Choose, seq:=Seq, 
CLUS pred:=Pred, rel:=Rel, id:=Id, uni:=Uni, incl:=Incl,
perms:=Perms, fsort:=FSort, sort:=Sort, combine:=Combine,
maxf:=Max, minf:=Min, max:=Max, min:=Min
"*:=ComposeF, "**":=ComposeR )

FUNCTION Union(s1, s2)->S = RET \{ t | s1(t) /\ s2(t) \} % s1 \ s2
FUNCTION Intersection(s1, s2)->S = RET \{ t | s1(t) /\ s2(t) \} % s1 \ s2
FUNCTION Difference(s1, s2)->S = RET \{ t | s1(t) \- s2(t) \} % s1 \- s2
FUNCTION In(s, t)->Bool = RET s(t) % t IN s
FUNCTION Subset(s1, s2)->Bool = RET (ALL t | s1(t) \-\ s2(t)) % s1 \ s2
FUNCTION Size(s)->Int = RET \{ t | s(t) \} => RET Size(s-{t}) + 1 [*] RET 0
FUNCTION 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.
FUNCTION 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
FUNCTION Pred(s)->(T->Bool) = RET s % s.pred
% s.pred is just s. We define pred for symmetry with seq.set, etc.
FUNCTION Rel(s)->(Bool->>T) = s % s.rel.inv
FUNCTION Incl(s)->(SET T->>T) = \{ st : SET T | t IN \ st \ / \ s \}.pToR
FUNCTION Perms(s)->SET SEQ T = RET s.seq.perm % s.perm
FUNCTION FSort(s, f: (T,T)->Bool)->S = RET s.seq.fsort(f) % s.sort(f);f is compare
FUNCTION Sort(s)->S = RET s.sort % s.sort; only if T has <=
FUNCTION Combine(s, f: (T,T)->>U)->T = RET s.seq.combine(f) % useful if f is commutative
FUNCTION FMax(s, f: (T,T)->Bool)->T = RET s.sort(f).last % s.max f; only if T has <=
FUNCTION FMin(s, f: (T,T)->Bool)->T = RET s.sort(f).head % s.min f; only if T has <=
FUNCTION Max(s)->T = RET s.max("<") % s.max; only if T has <=
FUNCTION Min(s)->T = RET s.min("<=") % 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. To get all the choices, use T.f.rel.leaves.
% Note that this is not the same as \ / : s, unless s is totally ordered.
FUNCTION ComposeF(s, f: T->U)->SET U = RET \{ t :IN s || f(t) \} % s * f; image of s under f
% ComposeF like sequences, pointwise on the elements. ComposeF(s, f) = ComposeR(s, f.rel)
FUNCTION ComposeR(s, r:T->U)->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 it explicitly:\{t :IN s, u | r(t, u) || u\}, or as \ / : (s * r=set.F).
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: T->U, 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 \rightarrow f2(t) \text{ [*]} f1(t) \text{ [*]} f(t) \text{ [*]} f1) \}

% (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 = (s.id * f).func

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

**FUNC** Normal(f, t)->Bool = t IN f.dom

**FUNC** Domain(f) -> SET T = f.rel.dom

**FUNC** Range (f) -> SET U = f.rel.rng

**FUNC** Rel(f) -> R = REL \{(t, u | f(t) = u).pToR \}

**FUNC** setRelRel(f, v: T) -> Bool = RET \{(t, v | (f!t \rightarrow v \text{ IN f(t) \text{ [*]} false} \text{ [*]} f(t)) \}

%If U = SET V, f.setRelRel relates each f in 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.

Functions declared with more than one argument take a single argument that is a tuple. So f(x: Int) takes an Int, but f(x: Int, y: Int) takes a tuple of type (Int, Int). This convention keeps the tuples in the background as much as possible. The normal syntax for calling a function is f(x, y), which constructs the tuple (x, y) and passes it to f. However, f(x) is treated differently, since it passes x to f, rather than the singleton tuple \{x}. If you have a tuple t
Sequences

A function is called a sequence if its domain is a finite set of consecutive \( \text{Int} \)'s starting at 0, that is, if it has type

\[ Q = \text{Int} \rightarrow T \text{ SUCHTHAT } (\forall q | (\exists\text{ size: Int } | q \text{. dom} = \{ 0..\text{size}.1 \}.\text{rng}) \]

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

- \( \text{head, tail, last, remi, addh, addl:} \) attaching or attaching the first or last element,
- \( \text{seg, sub:} \) extracting a segment of a sequence,
- \( +, \text{size:} \) concatenating two sequences, or finding the size,
- \( \text{fill:} \) making a sequence with all elements the same,
- \( \text{zip or ||:} \) making a pair of sequences into a sequence of pairs
-\( <-, <<: \) testing for prefix or sub-sequence (not necessarily contiguous),
-\( **: \) composing with a relation (\( \text{SEQ T} \) inherits composing with a function),
-\( \text{lexical comparison, permuting, and sorting,} \)
-\( \text{iterate, combine:} \) iterating a function over each prefix of a sequence, or the whole sequence 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 \( \text{addl} \) 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, \( \text{SEQ T} \text{ is Sequence[T].Q, where:} \)

\begin{verbatim}
MODULE Sequence[T] EXPORTS Q =

TYPE I = Int

Q = (I -> T) SUCHTHAT q.dom = (0 .. q.size-1).rng


% These methods treat a sequence as a multiset (or bag).

count:=Count, "IN":=In, "==":=EqElem, "\/:=":=Concatenate, "-":=Diff, set:=Q.rng }

FUNCTION Size(q) -> Int = q.dom.size

FUNCTION 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

FUNCTION Seg(q, i, n: I) -> Q = RET q.sub(i, i+n-1)

% q.seg(i,n) yields a sequence s from q(i)

FUNCTION Concatenate(q1, q2) -> Q = VAR q |

q.sub(0, q1.size-1) = q1 \ q.sub(q1.size, q.size-1) = q2 => RET q

FUNCTION Head(q) -> Q = RET q(0)

% q.head: first element
\end{verbatim}
A sequence is a special case of a tuple, in which all the elements have the same type.

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

There is a constructor \(\{e_1, e_2, \ldots\}\) 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 \(\{x : \text{IN } q | \text{pred1} \mid \text{exp}\}\) and \(\{x := e1 \text{ BY } e2 \text{ WHILE } \text{pred1} \mid \text{pred2} \mid \text{exp}\}\). For these see \([11]\) of section 5.

To map each element \(t\) of \(q\) to \(f(t)\) use function composition \(q * f.\) Thus if \(q: \text{SEQ Int,} q * (\{i: \text{IN } i \mid i*i\})\) yields a sequence of squares. You can also write this \(\{i: \text{IN } i \mid i*i\}\).
function constructor, 15, 24
function declaration, 16
function of type T whose value is result everywhere, 23
function undefined everywhere, 23
functional behavior, 2
general procedure, 2
global, 17, 18, 21
global, 31
GLOBAL.id, 17, 21
grammar, 2
graph, 24
greater or equal, 10
greater than, 10
greatest lower bound, 8
grouping, 16
guard, 4, 26, 24, 14, 15
Handler, 15
has a routine type, 4
has type T, 4
HAVOC, 15
head, 26
hierarchy, 31
history, 3, 7
id, 3
Id, 7
id := exp, 9
id [ typeList ], 5
identifier, 3
iff, 15
iff, 4, 26
IF, 15
IF, 28
if a then b, 7
implementer, 2
implication, 6, 7, 3
implies, 11, 10
IN, 11, 10, 22, 26
infinite, 3
infixOp, 10
initial value, 18
initialize, 16
instantiate, 17
Int, 9
intersection, 13, 10, 21
Introduction to Spec, 1
invocation, 26, 8, 11, 15
IS, 9
isPath, 25
join, 8
keyword, 3
known, 20
LAMBDA, 9, 12
lambda expression, 9
last, 26
lattice, 8
least upper bound, 8
less than, 10
lexical comparison, 20, 26
List, 3
literal, 3, 8, 9
local, 21
local, 4, 31
logical operators, 13
loop, 30
looping exception, 8, 14
m[typeList].id, 7
meaning
of an atomic command, 6
of an expression, 6
meaning of an atomic command, 14
meaning of an expression, 8
meet, 8
membership, 13, 10, 21
method, 4, 5, 6, 21
method, 8, 30
mf, 18
module, 2, 17, 18
module, 8, 31
monotonic, 8
multiply, 10
multiset, 20, 26
multiset difference, 10
name, 6, 1, 5, 8, 21
name space, 31
negation, 6
Nelson, 1
new variable, 16
non-atomic command, 6, 1, 14
non-atomic semantics, 7
Non-Atomic Semantics of Spec, 1
non-deterministic, 1
non-deterministic, 4, 5, 6, 28, 29
nonterminal symbol, 2
normal outcome, 6, 29
normal result, 23
not, 6
not equal, 11, 10
Null, 5
OD, 15
OD, 4, 30
only if, 7
operator, 3, 6
operator, 10
operators, 6
or, 6, 4, 28
ordering on Bool, 7
organizing your program, 7, 2
outcome, 14
outcome, 6
parameterized, 31
parameterized module, 17
path in the graph, 24
precedence, 10, 11, 6, 10, 15
precedence, 28
precedence, 30
precisely, 2
precondition, 15
pred, 9, 22
predefined identifiers, 3
predicate, 24
predicate, 3, 25, 26
Predicate logic, 8
prefix, 10, 20, 10, 26
prefixOp, 10
prefixOp e, 11
primary, 9
PROC, 7, 5, 18
procedure, 7
program, 2, 17, 18
program, 2, 4, 7
program counter, 7
propositions, 6
punctuation, 3
quantif, 9
quantification, 11
quantifier, 3, 4, 25
quantifiers, 9
quoted character, 3
RAISE, 9, 15
RAISE, 5
RAISE exception, 12
RAISES, 5, 12
RAISES, 5
RAISES set, 17
record, 5, 11
record constructor, 24
redeclaration, 20
Reflexive, 8
relation, 24
relation, 6
remh, 26
rem, 26
remove an element, 20, 26
removing an element, 13, 21
repetition, 30
result, 8
result type, 15
RET, 15
RET, 5
routine, 2, 15, 18
routine, 7
scope, 20
seg, 26
seq, 16
SEQ, 5, 6, 26
SEQ, 3
SEQ Char, 6
sequence, 28
sequence, 9, 30
sequence., 19, 26
sequential composition, 15
sequential program, 6, 1
set, 13, 11, 12, 21, 24
set, 3, 9
SET, 5
set constructor, 24
set difference, 10
set difference., 13, 21
set of sequences of states, 6, 1
set of values, 4
set with specific elements, 22
setF, 24
side effects, 16
side-effect, 8
signature, 16, 18
size, 26
SKIP, 15
Skolem function, 9
spec, 2
specification, 2, 4
specifications, 1
state, 1, 8, 14, 21
state, 2, 6
state machine, 1
state transition, 2
state variable, 6, 1
String, 5, 6
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 \([\text{name: String}, \text{salary: Int}]\) with \( \leq \) comparison of the \text{salary} field. Several \( T \)'s can have different \text{name}s but the same \text{salary}.

\[
\text{TYPE } S = \text{SET } T \\
Q = \text{SEQ } T
\]

APROC Sort(s) -> Q = <<
  VAR q | (ALL t | s.count(t) = q.count(t)) /\
  Sorted(q) => RET q >>

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

\[
\text{FUNC Sorted(q) -> Bool = RET } (\text{ALL i :IN q.dom - } \{0\} \mid q(i-1) < q(i))
\]

If we made \text{Sort} a FUNC rather than a PROC, what would be wrong? What could we change to make it a FUNC?

We could have written this more concisely as

\[
\text{APROC Sort(s) -> Q =}
  << \text{VAR q :IN a.perms | Sorted(q) => RET q >>}
\]

using the \text{perms} method for sets that returns a set of sequences that contains all the possible permutations of the set.

\footnote{Hint: a FUNC can't have side effects and must be deterministic (return the same value for the same arguments).}