

Contents

Preface	ix
1 Introducing abstract machines	1
1.1 Abstract machines	1
1.2 MACHINE	3
1.3 VARIABLES	4
1.4 INVARIANT	4
1.5 OPERATIONS	5
1.6 INITIALISATION	9
1.7 Exercises	9
2 Review of set theory and logic	11
2.1 Set theory	11
2.2 Logic notation	15
2.3 Substitution	21
2.4 Exercises	23
3 Weakest preconditions	25
3.1 State spaces	25
3.2 The notation $[S]P$	27
3.3 Laws of $[S]P$	30
3.4 Assignment	31
3.5 Multiple assignment	35

3.6	skip	37
3.7	Conditional	37
3.8	Case statement	40
3.9	BEGIN and END	41
3.10	Exercises	42
4	Towards machine consistency	45
4.1	Consistency of INVARIANT	45
4.2	Proof obligation for initialisation	46
4.3	Proof obligation for operations	47
4.4	Exercises	53
5	Parameters, sets, and constants	55
5.1	MACHINE parameters	56
5.2	CONSTRAINTS	57
5.3	SETS	58
5.4	CONSTANTS	58
5.5	PROPERTIES	59
5.6	Example: club behaviour	60
5.7	Full machine consistency	62
5.8	Summary	67
5.9	Exercises	67
6	Relations	69
6.1	Relations between sets	69
6.2	Domain restriction	72
6.3	Range restriction	72
6.4	Relational image	74
6.5	Relational inverse	75
6.6	Relational composition	75
6.7	Relations on a single set	76
6.8	Relational over-riding	79
6.9	Example: heirs to the throne	81

Contents	v
6.10 Relations in machines	85
6.11 Example: access	86
6.12 Exercises	88
7 Functions and sequences	91
7.1 Partial functions	91
7.2 Total functions	92
7.3 Injective functions	93
7.4 Surjective functions	94
7.5 Bijective functions	94
7.6 Lambda notation for functions	96
7.7 Example: tracking reading books	97
7.8 Sequences	98
7.9 Exercises	105
8 Arrays	107
8.1 Arrays as functions	107
8.2 Weakest preconditions for arrays	109
8.3 Exercises	117
9 Nondeterminism	119
9.1 Nondeterminism in specifications	119
9.2 The ANY statement	120
9.3 Nondeterministic assignment	126
9.4 The CHOICE statement	127
9.5 The SELECT statement	130
9.6 The PRE statement	133
9.7 Example: jukebox	135
9.8 Exercises	137
10 Structuring with INCLUDES	139
10.1 Inclusion	140
10.2 Included operations	143

10.3	Multiple inclusion	146
10.4	Parallel operations	148
10.5	Proof obligations	153
10.6	Exercises	156
11	Structuring with SEES and USES	159
11.1	The SEES relationship	159
11.2	Using SEES	161
11.3	The USES relationship	164
11.4	Proof obligations	165
11.5	Example: registrar	168
11.6	Exercises	171
12	Data refinement	173
12.1	Further AMN	173
12.2	Data refinement	176
12.3	Removing irrelevant information	184
12.4	Inheriting static information	187
12.5	Including and seeing machines in refinements	188
12.6	Example: connected towns	194
12.7	Summary: refinement machines	201
12.8	Exercises	203
13	Refinement of nondeterminism	205
13.1	Resolving nondeterminism	205
13.2	Relocating nondeterminism	209
13.3	Example: refining <i>Jukebox</i>	214
13.4	Exercises	216
14	Proof obligations for refinements	219
14.1	The <i>Colours</i> machine	219
14.2	Initialisation in refinement	219
14.3	Operations	224

Contents	vii
14.4 Operations with outputs	227
14.5 Proof obligations	228
14.6 Exercises	229
15 Loops	231
15.1 Loop execution	231
15.2 Loop invariant	233
15.3 Guaranteeing termination	236
15.4 Loop semantics	238
15.5 Loop development	242
15.6 Exercises	250
16 Implementation machines	255
16.1 IMPLEMENTATION	255
16.2 Example: a robust queue	264
16.3 Data refinement revisited	268
16.4 Loops in implementations	272
16.5 Implementing sets and constants	278
16.6 Exercises	283
17 Case study: heapsort	285
17.1 Priority queues and heaps	285
17.2 The loops	290
17.3 Sorting	293
17.4 Exercises	297
18 Library machines	299
18.1 Completing developments	299
18.2 A library machine: <i>Bool_TYPE</i>	301
18.3 Other TYPE machines	303
18.4 A library machine: <i>Nvar</i>	304
18.5 A library machine: <i>Varr</i>	308
18.6 Other simple state-maintaining machines	312

18.7	Collections	314
18.8	Example: supermarket checkouts	320
18.9	Other library machines	325
18.10	BASE machines	326
18.11	Exercises	336
	Answers to self tests	339
A	Generalised Substitution Language	357
B	Machine readable AMN	361
	Index	365
	Index of machines	369

Introducing abstract machines 1

A functional specification of a system component describes how it is required to behave. More specifically, it describes the interactions that the system offers to its user. It is important to know the operations that are offered by the system to the user, and what will be achieved when any of them is selected.

A specification can contain a significant amount of information. One of the difficulties in specification is managing the large volume of detailed system description that is required to formulate an accurate specification, and maintaining a high degree of confidence that it is all consistent. A structured approach to specification is essential for the production of any substantial system description.

The B-Method offers one such approach. The basic building block of a specification is the *abstract machine*. Large specifications can be constructed from smaller ones using a number of structuring mechanisms that permit a separation of concerns and support both comprehensibility and verification. The approach is *compositional*, in that the combination of abstract machines is again an abstract machine, permitting hierarchical specification. The notation that is used for writing such descriptions is called *Abstract Machine Notation*, abbreviated AMN. This chapter informally introduces some fundamental aspects of abstract machines, and the AMN notation for describing them.

1.1 Abstract machines

An abstract machine is a *specification* of a (part of a) system. It contains pieces of information that describe various aspects of the specification, and lists them under appropriate headings. An example is illustrated in Figure 1.1.

Firstly, any specification must describe what the component can do. The options offered to the user are called *operations*. These describe the functions that can be carried out by the component. They may take inputs from the user, supply outputs to the user, effect some change within the component, or any

```

MACHINE ...
VARIABLES ...
INVARIANT ...
INITIALISATION ...
OPERATIONS ...
END

```

Figure 1.1: Example headings within an abstract machine

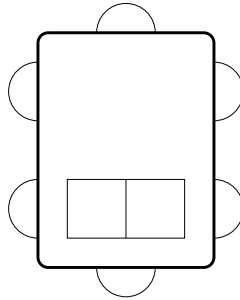


Figure 1.2: An abstract machine

combination of these. The collection of operations are the interface by which the machine interacts with its environment. These are all listed, along with a precise description of what they do, under the heading **OPERATIONS**.

A machine might also be required to maintain or process information. In this case it would be required to keep some local state. The state variables are listed under the **VARIABLES** heading. Their types, and any other information which concerns them, are listed under the **INVARIANT** heading. This contains all the information which must always be true of the state, whichever sequence of operations is selected. For example, the type of a variable should not change during the course of an execution, so it is given as part of the invariant.

The initial state should also be specified. This part of the specification appears under the **INITIALISATION** heading.

Finally, a machine must have a name in order to allow other parts of a large specification to refer to it. The name is given under the heading of **MACHINE**, which declares that the object being described is an abstract machine (rather than a refinement or an implementation, which will be covered in later chapters).

A machine is not unlike a description of an *object* in the object-oriented sense. It has a name, some internal state, and a set of operations, as do objects. It may be thought of as a black box with a set of buttons on the side (which can take inputs and provide outputs) corresponding to the operations, and a set of state variables inside. Interaction with the component must be through these

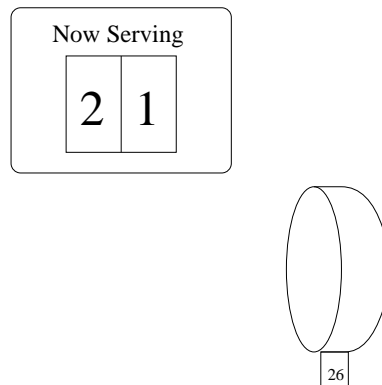


Figure 1.3: A ticketing system

operations, which comprise its interface. This view is illustrated in Figure 1.2. The state is maintained within the machine, and the buttons around the outside of the component represent the operations—the interface through which a user may interact with the machine.

To introduce the structure of a machine description in more detail, we will use an example of a ticket dispenser which is used in a shop to order the queue. On entry to the shop, a customer takes a numbered ticket from the dispenser. When a sales assistant is ready to serve, a display indicates the number of the customer who is to be served next. Customers wait until it is their turn to be served. The system is pictured in Figure 1.3.

The behaviour of this small system can be specified as an abstract machine. This will give details of the operations offered by the system, the internal state required to support those operations, and the additional information about that state.

1.2 MACHINE

The `MACHINE` clause is used to provide the name of the machine. All machines in a development must have different names. In the case of the ticket dispenser, we will call the machine *Ticket*. This part of the description is written as follows:

```
MACHINE Ticket
```

1.3 VARIABLES

An abstract machine is able to maintain some local state information in its variables. Since an abstract machine is concerned with specification, it is appropriate for these variables to be of the type that is most suitable for expressing the specification. These typically correspond to how systems are *understood*, rather than how they are implemented. This is generally in terms of values, sets, relations, functions, sequences, and other such constructs. A variable may be of type \mathbb{N} , the set of natural numbers $(0, 1, 2, \dots)$, but variables may also be of more abstract types such as relations or functions.

In the case of the *Ticket* machine, there are two pieces of information that it is important to maintain: the number of the next ticket to be dispensed, and the number of the ticket currently being served. These will be modelled by variables *next* and *serve* respectively.

All of the variables which are to be used in the machine are simply listed in the VARIABLES clause. Their types, as well as any other constraints, will be given in the INVARIANT clause. The VARIABLES clause in this case is written as

VARIABLES <i>serve, next</i>

1.4 INVARIANT

The INVARIANT clause provides all of the information about the variables of the machine. It must give all of their types, and it can also give other restrictions on their possible values, and their relationships to each other. The values of variables will change as the machine executes, but the invariant describes properties of the variables which must always be true as execution progresses. It also describes relationships between the variables and other parts of the system, as we shall see in later chapters.

Each variable must have its type given in the invariant, either as an element of a set (of the form $var \in TYPE$), or as a subset of a set (of the form $var \subseteq TYPE$)¹, or else as an equality (of the form $var = exp$). Hence there must be at least one clause of the invariant for each variable listed in the VARIABLES clause. In the case of the *Ticket* machine, both *next* and *serve* are keeping track of numbers, so they will both be of type \mathbb{N} , the set of natural numbers. Thus $next \in \mathbb{N}$ and $serve \in \mathbb{N}$ will appear in the INVARIANT clause of the machine.

The type information provides some information about how the variables are to be understood and used. Further information will impose further restrictions on what is understood by a sensible state of the system. In the *Ticket* machine, the number currently being served should never be greater than the ticket to be

¹This set notation is reviewed in Chapter 2

given out next. If the machine is able to reach a state in which $serve > next$ then something has gone wrong. The requirement that $serve \leq next$ must always be true simply reflects our understanding of the role that these two variables are playing in the specification, and so it should appear in the invariant. One major advantage of B is that it allows predicates such as $serve \leq next$ to be written in a machine processable way, something that ordinary type systems cannot do.

The INVARIANT clause is an *assertion*, or a claim about the reachable states of the machine. The INVARIANT clause characterises the sensible states that are permitted for the machine. The machine should never arrive at a state in which some part of the INVARIANT clause is false. The complete invariant for the *Ticket* machine is therefore the conjunction of the type information and the other constraint:

INVARIANT $serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next$
--

1.5 OPERATIONS

The OPERATIONS clause of the machine description contains a list of operation definitions.

In software engineering, specifications of operations generally provide the following information:

- The name of the operation;
- Input parameters;
- Output parameters;
- What the operation requires (restrictions on parameters and the states from which the operation may be called);
- What the operation modifies (variables that may be modified);
- The effects or behaviour of the operation (what the operation does).

AB description of an operation provides all of this information, in a form which is suited to structuring specifications.

The name, and input and output parameters of an operation are given by an operation header:

$$outputs \leftarrow name(inputs)$$

where *name* is the name of the operation, *outputs* is a list of output variables, and *inputs* is a list of input variables. The variables are all formal parameters, so they must all be different.

The *name* of the operation must be given, but both the input list and the output list are optional, since some operations do not take inputs, and some do not provide outputs.

The *Ticket* machine will offer two operations, one to serve the next customer (and update the indicator board), and one to provide a customer with the next ticket. These are declared as follows²:

$$\begin{aligned} ss &\leftarrow \mathbf{serve_next} \\ tt &\leftarrow \mathbf{take_ticket} \end{aligned}$$

Neither of them takes any input, and they both provide a single output: *ss* in the case of **serve_next**, and *tt* in the case of **take_ticket**.

The operation specification itself consists of a *precondition* part, and a *body*. The precondition must contain type information for all of the input variables, as well as any further assumptions on the inputs or the state of the machine. These together describe what the operation requires in order to be sure to behave correctly. The precondition therefore corresponds to a requirement on the user to ensure that the requirements are met whenever the operation is called.

For example, the **serve_next** operation is likely to be called when the sales assistant has finished serving a customer. In this case, *serve* (the ticket number of the person who has just finished being served) must be strictly less than *next*, the number on the next ticket to be dispensed. If the shop is empty, then the value of *serve* will be the same as *next*. In this situation, the **serve_next** operation should not be called before a ticket is taken. Thus the precondition for this operation will be $serve < next$.

The body of an operation describes what the operation achieves. It must assign some value to each of the output variables, and it can also update the machine state. The effect of an operation is therefore specified by giving the relationship between the state before it is called together with the input values, and the state after the operation has completed together with the output values. This is accomplished in B by giving an abstract assignment statement which determines how the state is to be updated and what the output values should be in terms of the initial state and input values.

In the case of **serve_next**, the state variable *serve* should be incremented, and the output (to the display board) should also be incremented to mirror the value of the state *serve*. In AMN, an assignment is written as $x := E$ (pronounced 'x becomes E') where *x* is the variable to be assigned, and *E* is the expression whose value is to be assigned to *x*. The variable *serve* is thus incremented by the assignment $serve := serve + 1$.

²The B-Toolkit requires any variable name to be at least two characters long. Thus double-letter variable names such as *ss* and *tt* often appear in B descriptions

The output ss of this operation is to be assigned the same value. This is achieved by $ss := serve + 1$. Since the operation is required to achieve both of these assignments, it may be described by means of a *multiple assignment* which simultaneously assigns to both ss and $serve$:

$$ss, serve := serve + 1, serve + 1$$

Since the body of an operation is intended to act as a *specification*, it should relate the states of the machine before and after the operation in as clear and transparent a way as possible. For this reason, such specifications cannot have intermediate states, so multiple assignments have to be described as a simultaneous assignment, rather than a sequence of assignments one after the other.

The complete specification of the operation **serve_next** is as follows:

```

ss ← serve_next ≐
  PRE   $serve < next$ 
  THEN  $ss, serve := serve + 1, serve + 1$ 
  END

```

The type of the output variable ss is given by the assignment, so output types do not need to be declared separately: they are completely determined from the body of the operation.

Self Test 1.1 Specify the operation **serve_two**, which serves the next two people in the queue. □

For an operation to be a consistent part of a machine description, it must preserve the invariant of the machine whenever it is invoked legitimately, with the precondition true. In other words, if the precondition is true of the state and inputs of the operation, then the body of the operation must guarantee that the invariant is still true on the updated state.

There are no consistency requirements on the values that are output. The invariant is concerned only with legitimate machine states, so it cannot constrain the outputs of an operation.

In the case of **serve_next**, provided the user only invokes it when the precondition $serve < next$ is true, the resulting update of the state will indeed meet the invariant $serve \leq next$, since $serve$ is only ever increased by 1. Hence this operation is consistent with the invariant. The value of ss that is output cannot be mentioned by the invariant—it is local to the operation which declares it. The assignment to ss simply specifies the output that is required from the operation.

If the precondition of **serve_next** was weaker, so it allowed more flexibility to the user (by imposing fewer conditions on when the operation can legitimately

be called), then the operation might no longer be consistent with its invariant. Consider the situation where the operation can *always* be called, whatever state the machine is in. This corresponds to a precondition of *true*, with a resulting operation

```

ss ← serve_next  $\hat{=}$ 
  PRE true
  THEN ss, serve := serve + 1, serve + 1
  END

```

In this case, the operation can be invoked even when *serve* = *next*, and will reach a state where the invariant is no longer true. This weaker version of the operation is no longer consistent with the invariant, since it allows the user to fulfil his or her own obligations but does not meet its own. The original version of **serve_next** can violate the invariant only if the user fails to meet the requirements on executing the operation.

Strengthening the precondition of **serve_next** to *serve* < *next* imposes more constraints on the user, and in this case results in the operation becoming consistent. There is a natural tension between usability (which needs the preconditions to be as weak as possible) and correctness (which needs the preconditions to be restrictive enough to prevent operations being called when they should not be). In the extreme case, an operation can have a precondition which can never be fulfilled. In this case, it is correct in that it can never bring the machine to an invalid state, but at the expense of being completely unusable.

The operation **take_ticket** can never result in an incorrect state, since it increments *next* (outputting the current value) and leaves *serve* unchanged. The complete operation is specified as follows:

```

tt ← take_ticket  $\hat{=}$ 
  PRE true
  THEN tt, next := next, next + 1
  END

```

It has the weakest possible precondition—*true*—indicating that there are no constraints on the user regarding its invocation.

When a precondition is *true*, then it may be dropped from the specification of its operation. A shorter way of specifying **take_ticket** is as follows:

```

tt ← take_ticket  $\hat{=}$ 
  tt, next := next, next + 1

```

If no precondition is given, then it is assumed to be *true*. If the operation is consistent with this precondition, then it must still be correct with any stronger one.

Self Test 1.2 Suppose that the invariant has a further clause $next \leq serve + 20$ indicating that there must never be more than 20 outstanding tickets. Why is the operation **take_ticket** not consistent with this invariant? How should the precondition of **take_ticket** be strengthened to preserve its consistency with the invariant? \square

Self Test 1.3 Is the operation

$$tt \leftarrow \text{replace_ticket} \hat{=} next := next - 1$$

consistent with the invariant of the *Ticket* machine? \square

1.6 INITIALISATION

The INITIALISATION clause is used to describe the possible initial state of the machine. It consists of an AMN statement which is used to set the state in which the machine starts. All variables listed in the VARIABLES clause must be assigned some value.

The *Ticket* machine should start with 0 on the display board, and 0 as the number of the first ticket to be taken. Hence both variables should be set to 0, so a multiple assignment will be used to set them both simultaneously. The INITIALISATION clause is thus written as follows:

INITIALISATION $serve, next := 0, 0$

For the INITIALISATION clause to be a consistent part of the specification, possible initial states must be correct with respect to the invariant. This means that the program in the INITIALISATION clause must be guaranteed to establish the invariant.

The complete specification of the *Ticket* machine is given in Figure 1.4.

1.7 Exercises

Exercise 1.1 Add a **reset** operation to the *Ticket* machine, which returns it to its initial state. \square

Exercise 1.2 Amend the description of the *Ticket* machine so that only tickets numbered up to 500 can be dispensed. (This will involve a change to the invariant of the machine, and to the operation **take_ticket**.) \square

```

MACHINE Ticket
VARIABLES serve, next
INVARIANT  $serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next$ 
INITIALISATION  $serve, next := 0, 0$ 
OPERATIONS
  ss  $\leftarrow$  serve_next  $\hat{=}$ 
    PRE  $serve < next$ 
    THEN  $ss, serve := serve + 1, serve + 1$ 
    END ;
  tt  $\leftarrow$  take_ticket  $\hat{=}$ 
    PRE true
    THEN  $tt, next := next, next + 1$ 
    END
END

```

Figure 1.4: The *Ticket* machine

Exercise 1.3 Add a **query** operation which outputs the number of people currently waiting in the queue. \square

Exercise 1.4 Add another state variable, *record*, to the machine. This variable should keep track of the number of people that have taken a ticket since the last **serve_next** operation. It will be incremented when **take_ticket** occurs, and reset to 0 when **serve_next** occurs. You should also add a query operation which outputs the current value of *record*. \square

Exercise 1.5 Adapt the *Ticket* machine so that you can add a **limit** operation, which takes a number *nn* as input, and will not allow the total number of tickets issued (i.e. the value of *next*) to be greater than this number. (You will have to add another state variable.) The initial limit should be 500. \square

Exercise 1.6 Is the operation: **undo_serve** $\hat{=}$ $serve := serve - 1$ consistent with the invariant of the *Ticket* machine? \square

Exercise 1.7 A car park has 640 parking spaces. Give an abstract machine which specifies a system to control cars entering the car park. It should keep track of the number of cars currently in the car park, and should provide three operations:

- **enter**, which records the entry of a new car. This should occur only when the car park is not full;
- **leave**, which records the exit of a car from the car park;
- $nn \leftarrow$ **query**, which outputs the number of cars currently in the car park.

 \square

Index

- abstract machine, *see* MACHINE
- Abstract Machine Notation, *see* AMN
- AMN, ix, 1, 362
- ANY statement, 120
 - multiple variables, 125
 - weakest precondition, 123, 126
- append to sequence (\leftarrow), 100
- arithmetic notation, 363
- array assignment, 109
 - multiple updates, 113
 - weakest precondition, 109
- ascii notation, 361-364
- assertion, 5, 15
- assignment, 31
 - parallel, 36
- assignment statement, 6, 357
 - weakest precondition, 31

- BASE, 327
- BASE machines, 326-336
- BEGIN...END, 41
- bijective function, 94
- bool, 302
- Bool_TYPE*, 301
- bound variable, 20
- bounded choice statement, 357

- cardinality (card), 14
- cartesian product (\times), 13, 69
- CASE statement, 40
 - weakest precondition, 41
- CHOICE statement, 127
 - weakest precondition, 129

- closure (relational), 78
- composition
 - functional, 97
 - relational, 75
- compositionality, 1, 38
- concatenation ($\hat{\ }^{\circ}$), 100
- conditional statement, *see* IF
- conjunction, 16
- consistency, 45
 - of operations, 7
- constants
 - deferred, 55
 - implementation, 280
- CONSTANTS, 58
- CONSTRAINTS, 57
 - proof obligations, 63

- data refinement, 176
 - in implementations, 268
- disjoint, 13
- disjunction, 15
- domain anti-restriction
 - functional, 97
 - relational, 72
- domain of relation, 71
- domain restriction
 - functional, 97
 - relational, 72

- empty sequence ($[]$), 99
- empty set, 12
- enumerated set, 58
- equivalence class, 78, 194

- equivalence relation, 78
- Euclid's algorithm, 249
- existential quantification, 18
- expression, 6, 21
- EXTENDS, 143
- feasible, 29
- final segment of sequence (\downarrow), 101
- finite power set (\mathbb{F}), 313
- first (of sequence), 100
- free variable, 20
- front (of sequence), 100
- full-hiding, 256
- function notation, 96, 364
 - bijection (\leftrightarrow), 95
 - injection
 - partial (\rightarrow), 93
 - total (\twoheadrightarrow), 93
 - lambda (λ), 96
 - partial (\dashrightarrow), 92
 - surjection
 - partial (\twoheadrightarrow), 94
 - total (\rightarrow), 94
 - total (\rightarrow), 92
- Galler-Fischer, 194
- Generalised Substitution Language,
 - see* GSL
- GSL, xiii, 357
- guarded statement, 132, 358
- heapsort, 285–296
- identity relation, 76
- if and only if, 17
- IF statement, 37
 - weakest precondition, 37, 38
- image (relational), 74
- IMPLEMENTATION, 256
 - initialisation, 259
 - proof obligations, 256
 - restricted language, 258
- implication, 16
- IMPORTS, 256
- included machine initialisation, 141
- INCLUDES, 140
 - proof obligations, 154–155
 - transitivity, 147
- infeasible, 29
- initial segment of sequence (\uparrow), 101
- INITIALISATION, 9
 - proof obligations, 46, 65
- injective
 - function, 93
 - sequence, 102
- intersection (\cap), 12
 - general (\bigcap), 13
- INVARIANT, 4
 - of a loop, 236
 - proof obligations, 45, 65
- inverse
 - functional, 97
 - relational, 75
- iseq (injective sequences), 101
- iteration (relational), 78
- lambda (λ) abstraction, 96
- last (of sequence), 100
- laws of parallel reductions, 148
- laws of relations, 73, 75, 76
- laws of weakest preconditions, 30
- layered development, ix, 259, 299
- LET statement, 124
 - weakest precondition, 125
- library machines, 299–326
- linking invariant, 177, 207, 221
- list (sequence), 99
- local variables, 124, 175
- logic notation, 20, 361
 - and (\wedge), 16
 - exists (\exists), 18
 - for all (\forall), 18
 - if and only if (\Leftrightarrow), 17
 - implication (\Rightarrow), 16
 - negation, 16
 - not (\neg), 16
 - or (\vee), 15
- logical equivalence, 17
- loop
 - invariant, 233
 - nested, 275
 - variant, 236
- loop statement, *see* WHILE

- MACHINE, 3
 - proof obligations, 46, 47, 67
 - template, 63
- machine readable notation, 361-364
- MANDATORY, 326
- maps to, 69
- max, 127
- min, 205
- miracle, 358
- mod, 363
- multiple assignment, 7, 35
 - weakest precondition, 35, 36
- multiple object machines, 314-320
- multiple substitution, 22

- native information, 140
- natural numbers (\mathbb{N}), 11
- negation, 16
- non-termination, 29
- nondeterminism, 29
- nondeterministic assignment, 126
 - weakest precondition, 126
- normalised form, 358

- object, 2
- operations, 1
- OPERATIONS, 5
 - inconsistency, 49
 - proof obligations, 48, 66
- OPTIONAL, 326
- ordered pair (s, t) , 13, 69
- over-riding
 - functional, 97
 - relational, 79

- parallel assignment, 36
- parallel composition, 148-149, 359
- parameters
 - formal and actual, 143
 - of imported machine, 256
 - of included machine, 140
 - of MACHINE, 56
 - of operations, 5, 143
- partial function, 91
- perm (permutations), 101
- Pi (Π), 244

- positive numbers (\mathbb{N}_1), 14
- postcondition, 27
- power set (\mathbb{P}), 13
- PRE statement, 133
 - weakest precondition, 134
- precondition, 27, 358
- predicate, 15
- prefix to sequence (\rightarrow), 100
- priority queue, 285
- product, 244
- PROMOTES, 142
- PROPERTIES, 59
 - proof obligations, 64

- quantification, 18
 - over variable list, 18
- query operations, 50

- range anti-restriction
 - functional, 97
 - relational, 73
- range of relation, 71
- range restriction
 - functional, 97
 - relational, 72
- refinement
 - and structuring mechanisms, 188-194
 - common variables, 205
 - relocating nondeterminism, 209-213
 - removing irrelevant information, 184-186
 - resolving nondeterminism, 205-209
 - reusing static information, 187-188
- REFINEMENT, 178
 - proof obligations, 228-229
 - operations, 222, 224
 - output operations, 227
 - preconditions, 226
 - relationship to MACHINE, 179
 - template, 201
- reflexive relation, 76
- relation definition, 70

- relation notation, 363
 - closure (R^*), 78
 - composition ($;$), 76
 - domain anti-restriction (\Leftarrow), 72
 - domain restriction (\triangleleft), 72
 - identity relation ($id(S)$), 76
 - image ($R[U]$), 74
 - inverse (R^{-1}), 75
 - iteration (R^n), 78
 - non-reflexive closure (R^+), 79
 - over-riding (\Leftarrow), 79
 - range anti-restriction (\triangleright), 73
 - range restriction (\triangleright), 72
 - relation (\leftrightarrow), 70
- reverse (rev) of sequence, 100
- Russian Multiplication, 237
- scope, 20
- SEES, 159
 - proof obligations, 166-167
- SELECT statement, 130
 - weakest precondition, 132
- semi-hiding, 139
- seq, 101
- seq₁ (non-empty sequences), 101
- sequence notation, 104, 364
- sequence of numbers, 12
- sequences, 98
- sequential composition, 173
 - weakest precondition, 174
- set comprehension, 12
- set definition, 11, 12
- set membership (\in), 11
- set names, 11
- set non-membership (\notin), 11
- set notation, 14, 362
- set subtraction ($-$), 13
- sets, 11
 - deferred, 55, 278
 - implementation, 278
- SETS, 58
- Sigma (Σ), 112
- size (of sequence), 99
- size of set, 14
- skip, 37
 - weakest precondition, 37
- sorting algorithm, 285
- source of relation, 70
- specification, 1
- state space, 25
- state transformation, 25
- subset (\subseteq), 12
- substitution, 19, 21
 - multiple, 22
- substitution statement, *see* assignment statement
- sum, 112
- surjective function, 94
- symmetric relation, 77
- SYSTEM, 326
- tail (of sequence), 100
- target of relation, 70
- total function, 92
- transition, 26
- transitive relation, 77, 78
- truth tables, 15-16
- TYPE machines, 301-304
- unbounded choice statement, 358
- underspecification, 29
- union (\cup), 12
 - general (\cup), 12
- universal quantification, 18
- USES, 165
- VAR statement, 175
 - weakest precondition, 176
- variable capture, 20, 22
- variable names, 6
- variables
 - in machines, 2
 - in predicates, 18
- VARIABLES, 4
- VARIANT, 237
- weakest precondition, 27
- WHILE statement, 232
 - proof obligations, 238
 - invariant, 234
 - termination, 234
 - variant, 237
 - weakest precondition, 238

Index of machines

Access, 87
Allocate, 206
AllocateR, 207
Archive, 262
Array, 294

Baskets, 338
Body, 274
Books, 210
BooksR, 210
BooksRR, 213
Bool_TYPE, 302
Bool_TYPE_Ops, 302
Booth, 218
Bus, 54

Checkouts, 321
CheckoutsI, 323
Club, 62
Colours, 220
Colours2, 230
ColoursR, 220
ColoursR2, 229
Counter, 266
Customer, 164
CustomerI, 260

Date, 283
Doors, 145

Exam, 185
ExamR, 186

Fifo, 261

FifoI, 263
FifoI2, 271

Garray, 277
Goods, 162

Heaparray, 291, 292
HeapI, 288, 289
Hotel, 115
Hotelguests, 117
HotelguestsI, 311

Ironing, 188
IroningR, 189

Jobshop, 217
Jukebox, 135, 136
JukeboxR, 215

Keys, 151

Life, 168
List, 191
Locks, 146

Map, 192
Marks, 337
Marriage, 169
Mult, 273
Multi, 274

Names, 278
NamesI, 278
Nvar, 305

Paperround, 52
Pfun, 324
Player, 204
Port, 190
PortR, 192
PositionCounter, 272
PositionCounterI, 307
Price, 163
Priorityqueue, 286
Projects, 334
ProjectsDataBase, 328, 330, 332, 333
ProjectsI, 335

Reading, 98, 99
Registrar, 171
Results, 103
RobustFifo, 265
RobustFifoI, 267
Rooms, 281
RoomsI, 282

Safes, 152

seq_ctx, 315
seq_obj, 314, 317, 319
Set, 260
Shop, 164
SizeCounter, 272
Sortarray, 294
SortarrayI, 295

Team, 178
TeamR, 179, 184
Ticket, 10
Time, 279
Timefields, 279
TimefieldsI, 280
TimeI, 279
Towns, 195
TownsR, 197
TownsRR, 198
TownsRRR, 200
TownsRRRI, 276

Varr, 309
Varray, 271