James Clerk Maxwell Building
University of Edinburgh
Mayfield Road
Edinburgh EH9 3JZ, Scotland
Great Britain

AMS Subject Classifications (1970): 68-A30

Library of Congress Cataloging in Publication Data

Gordon, Michael J C 1948–
The denotational description of programming languages.

Bibliography: p.
Includes indexes.
1. Programming languages (Electronic computers)
I. Title.
QA76.7.G67 001.6'424 79-15723

All rights reserved.

No part of this book may be translated or reproduced
in any form without written permission from Springer-Verlag.

© 1979 by Springer-Verlag New York Inc.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-90433-6 Springer-Verlag New York Heidelberg Berlin
ISBN 3-540-90433-6 Springer-Verlag Berlin Heidelberg New York

Preface

This book explains how to formally describe programming languages using the techniques of *denotational semantics*. The presentation is designed primarily for computer science students rather than for (say) mathematicians. *No* knowledge of the theory of computation is required, but it would help to have some acquaintance with high level programming languages. The selection of material is based on an undergraduate semantics course taught at Edinburgh University for the last few years. Enough descriptive techniques are covered to handle all of ALGOL 60, PASCAL and other similar languages.

Denotational semantics combines a powerful and lucid descriptive notation (due mainly to Strachey) with an elegant and rigorous theory (due to Scott). This book provides an introduction to the descriptive techniques without going into the background mathematics at all. In some ways this is very unsatisfactory; reliable reasoning about semantics (e.g. correctness proofs) cannot be done without knowing the underlying model and so learning semantic notation without its model theory could be argued to be pointless. My own feeling is that there is plenty to be gained from acquiring a purely intuitive understanding of semantic concepts together with manipulative competence in the notation. For these equip one with a powerful conceptual framework—a framework enabling one to visualize languages and constructs in an elegant and machine-independent way. Perhaps a good analogy is with calculus: for many practical purposes (e.g. engineering calculations) an intuitive understanding of how to differentiate and integrate is all that is needed. Once the utility of the ideas and techniques are appreciated it becomes much easier to motivate the underlying mathematical notions (like limits and continuity). Similarly an intuitive understanding of the descriptive techniques of denotational semantics is valuable, both as a tool for understanding programming, and as a motivation for the advanced theory.

Because the underlying mathematics is not described I have occasionally used notation which, whilst intuitively straightforward, is technically sloppy. For example I have used the symbol = with several conceptually similar, but mathematically distinct, meanings. I felt it best not to

2. A first example: the language TINY

In this chapter we shall describe the syntax and semantics of a little programming language called TINY. Our purpose is to provide a vehicle for illustrating various formal concepts in use. In subsequent chapters we shall describe these concepts systematically but here we just sketch out the main ideas and associated techniques.

2.1. Informal syntax of TINY

Tiny has two main kinds of constructs, *expressions* and *commands*, both of which can contain *identifiers* which are strings of letters or digits beginning with a letter (for example **x, y1, thisisaverylongidentifier**). If we let **I**, $\mathbf{I_1}$, $\mathbf{I_2}$, . . . stand for arbitrary identifiers, **E**, $\mathbf{E_1}$, $\mathbf{E_2}$, . . . stand for arbitrary expressions and **C**, $\mathbf{C_1}$, $\mathbf{C_2}$, . . . stand for arbitrary commands then the constructs of TINY can be listed as follows:

E::=0 | 1 | true | false | read | I | not E | $\mathbf{E_1}$ = $\mathbf{E_2}$ | $\mathbf{E_1}$ + $\mathbf{E_2}$
C::=I:=E | output E | if E then $\mathbf{C_1}$ else $\mathbf{C_2}$ | while E do C | $\mathbf{C_1}$;$\mathbf{C_2}$

This notation is a varient of BNF, the symbol "::=" should be read as "can be" and the symbol "|" as "or". Thus a command **C** can be **I:=E** or **output E** or **if E then $\mathbf{C_1}$ else $\mathbf{C_2}$** or **while E do C** or **$\mathbf{C_1}$;$\mathbf{C_2}$**. Notice that this syntactic description is ambiguous—for example it does not say whether **while E do $\mathbf{C_1}$;$\mathbf{C_2}$** is **(while E do $\mathbf{C_1}$);$\mathbf{C_2}$** or **while E do ($\mathbf{C_1}$;$\mathbf{C_2}$)**. We shall use brackets (as above) and indentation to avoid such ambiguities. In the next chapter we shall clarify further our approach to syntax.

2.2 Informal semantics of TINY

Each command of TINY, when executed, changes the *state*. This state has three components:

(i) The *memory*: this is a correspondence between identifiers and *values*. In the memory each identifier is either *bound* to some value or *unbound*.

(ii) The *input*: this is supplied by the user before programs are run; it consists of a (possibly empty) sequence of values which can be read using the expression **read** (explained later).

(iii) The *output*: this is an initially empty sequence of values which records the results of the command **output E** (explained later).

Each expression of TINY specifies a value; since expressions may contain identifiers (for example **x + y**) this value depends on the state. All values—the values of expressions, the values bound to identifiers or the values in the input or output—are either truth values (**true**, **false**) or numbers (**0,1,2,...**).

We shall now explain informally the meaning of each construct.

2.2.1. Informal semantics of expressions

The value of each expression is as follows:

(E1) **0** or **1**
The value of **0** is the number **0** and the value of **1** is the number **1**.

(E2) **true** or **false**
The value of **true** is the truth value **true** and the value of **false** is the truth value **false**.

(E3) **read**
The value of **read** is the next item on the input (an error occurs if the input is empty). **read** has the 'side effect' of removing the first item so after it has been evaluated the input is one item shorter.

(E4) **I**
The value of an expression **I** is the value bound to **I** in the memory (if **I** is unbound an error occurs).

(E5) **not E**
If the value of **E** is **true** then the value of **not E** is **false**; if the value of **E** is **false** then the value of **not E** is **true**. In all other cases an error occurs.

(E6) $\mathbf{E_1} = \mathbf{E_2}$
The value of $\mathbf{E_1} = \mathbf{E_2}$ is **true** if the value of $\mathbf{E_1}$ equals the value of $\mathbf{E_2}$, otherwise it is **false**.

(E7) $\mathbf{E_1} + \mathbf{E_2}$
The value of $\mathbf{E_1} + \mathbf{E_2}$ is the numerical sum of the values of $\mathbf{E_1}$ and $\mathbf{E_2}$ (if either of these values is not a number then an error occurs).

2.2.2. Informal semantics of commands

(C1) **I**:=**E**

I is bound to the value of **E** in the memory (overwriting whatever was previously bound to **I**).

(C2) **output E**

The value of **E** is put onto the output.

(C3) **if E then** $\mathbf{C}_1$ **else** $\mathbf{C}_2$

If the value of **E** is **true** then $\mathbf{C}_1$ is done, if its value is **false** then $\mathbf{C}_2$ is done (in any other case an error occurs).

(C4) **while E do C**

If the value of **E** is **true** then **C** is done and then **while E do C** is repeated starting with the state resulting from **C**'s execution. If the value of **E** is **false** then nothing is done. If the value of **E** is neither **true** nor **false** an error occurs.

(C5) $\mathbf{C}_1;\mathbf{C}_2$

$\mathbf{C}_1$ and then $\mathbf{C}_2$ are done in that order.

2.3. An example

The following TINY command outputs the sum of the numbers on the input. The end of the input is marked with **true**.

```
sum:=0; x:=read;
while not (x=true) do sum:=sum+x; x:=read;
output sum
```

2.4. Formal semantics of TINY

We shall now informally formalize the above description of TINY. Our hope is to convey the general 'shape' of a denotational description. The reader should not attempt to grasp all the details (some of which are over-simplified) but just get the main ideas.

An essential part of our formalization will be the defining of various sets (for example sets of denotations). It turns out that some of the ways of defining sets we need only work properly if we use certain special sets called *domains*. We shall thus use "domain" instead of "set"—however intuitively domains can be thought of just like sets (indeed the class of

domains is just a subclass of the class of sets) and we shall employ normal set theoretic notation on them. For example $\{\mathbf{x} \mid P[\mathbf{x}]\}$ is the set of all **x**'s satisfying $P[\mathbf{x}]$; $\mathbf{x}\varepsilon\mathbf{S}$ means **x** belongs to **S**; $\mathbf{f}:\mathbf{S}_1\rightarrow\mathbf{S}_2$ means **f** is a function from $\mathbf{S}_1$ to $\mathbf{S}_2$. In the next chapter (in fact in 3.2.) we shall explain why domains rather than sets must be used.

2.4.1. Syntax

To deal with the syntax we define the following *syntactic domains*:

Ide = {**I** | **I** is an identifier}
Exp = {**E** | **E** is an expression}
Com = {**C** | **C** is a command}

Domain names like these will always start with a capital letter. **Ide** is a standard domain, **Exp** and **Com** vary from language to language.

2.4.2. States, memories, inputs, outputs and values

We start by formalizing the concept of a state. To do this we define domains **State** of states, **Memory** of memories, **Input** of inputs, **Output** of outputs and **Value** of values. The definition of these domains consists of the following *domain equations* which we first state and then explain:

(i) **State** = **Memory** × **Input** × **Output**
(ii) **Memory** = **Ide**→[**Value** + {**unbound**}]
(iii) **Input** = **Value***
(iv) **Output** = **Value***
(v) **Value** = **Num** + **Bool**

(i) Means that **State** is the domain of all triples **(m,i,o)** where $\mathbf{m}\varepsilon$ **Memory**, $\mathbf{i}\varepsilon$**Input** and $\mathbf{o}\varepsilon$**Output**. In general $\mathbf{D}_1 \times \mathbf{D}_2 \times \ldots \times \mathbf{D}_n$ is the domain $\{(\mathbf{d}_1,\mathbf{d}_2,\ldots,\mathbf{d}_n) \mid \mathbf{d}_1\varepsilon\mathbf{D}_1, \mathbf{d}_2\varepsilon\mathbf{D}_2, \ldots, \mathbf{d}_n\varepsilon\mathbf{D}_n\}$ of n-tuples.

(ii) Means that **Memory** is the domain of all functions from the domain **Ide** to the domain [**Value**+{**unbound**}]. The domain [**Value**+{**unbound**}] is the union of the domain **Value** and the one element domain {**unbound**}. In general $[\mathbf{D}_1\rightarrow\mathbf{D}_2]$ is the domain $\{\mathbf{f} \mid \mathbf{f}:\mathbf{D}_1\rightarrow\mathbf{D}_2\}$ of all functions from $\mathbf{D}_1$ to $\mathbf{D}_2$ and $[\mathbf{D}_1 + \mathbf{D}_2]$ is the 'disjoint union' of $\mathbf{D}_1$ and $\mathbf{D}_2$—we clarify this later; for now think of + as ordinary union restricted to disjoint domains. If $\mathbf{m}\varepsilon$**Memory** and $\mathbf{I}\varepsilon$**Ide** then **m I**, the result of applying the function **m** to argument **I**, is either in **Value** or is **unbound**; in the former case the

value **m I** is the value bound to **I** in **m**, in the latter case **I** is unbound in **m**.

(iii) Means **Input** is the domain of all strings (including the empty string) of members of **Value**. In general **D*** is the domain

$$\{(d_1,\ldots,d_n) \mid d_1 \varepsilon D, d_2 \varepsilon D, \ldots, d_n \varepsilon D\}$$

of strings or sequences over **D**. We shall denote the empty string by **( )** and sometimes write $d_1 \,.\, d_2 \,.\, \ldots \,.\, d_n$ instead of $(d_1,\ldots,d_n)$.

(iv) Means **Output** is the domain of all strings of values.

(v) Means that a value is either a number (i.e. a member of **Num**) or a truth value (i.e. a member of **Bool**). **Num** = {0,1,2,...} and **Bool** = {**true, false**}. Thus a state is a triple **(m,i,o)** where **m** is a function from identifiers to values (or **unbound**) and **i** and **o** are sequences of values.

2.4.3. Semantic functions

In this section we discuss the *semantic functions* for TINY. Semantic functions are functions which define the denotation of constructs. For TINY we need:

E:**Exp**→{denotations of expressions}
C:**Com**→{denotations of commands}

If **E**ε**Exp** and **C**ε**Com** then **E**[**E**] and **C**[**C**] are the results of applying the functions **E** and **C** to **E** and **C** respectively and are the denotation of the corresponding constructs defined by the semantics. We discuss these denotations later, but first note that:

(i) In general if **X** is a variable which ranges over some syntactic domain of constructs then we will use **X** for the corresponding semantic functions. Thus **C**[**I**:=**E**] is the denotation—or meaning—of **I**:=**E** etc.

(ii) The "emphatic brackets" [and] are used to surround syntactic objects when applying semantic functions to them. They are supposed to increase readability but have no other significance—**X**[**X**] is just the result of applying the function **X** to **X**.

2.4.3.1. Denotations of expressions

Since expressions produce values one might at first take their denotations to be members of **Value**. To model this idea one would give the semantic function **E** the type **Exp**→**Value** and then **E**[**E**] would be E's value (e.g. **E**[**0**]=**0**). This works for constant expressions but in general it fails to handle:

(ii) The dependence of some expression's values on the state (for example the value of **x**+**1** depends on what **x** is bound to in the memory; the value of **read** depends on the input).

(iii) The possibility that the evaluation of an expression might change the state (for example **read** removes the first item from the input).

To handle (i) we must define **E**:**Exp**→[**Value**+{**error**}] so that:

$$\mathbf{E}[\mathbf{E}] = \begin{cases} \mathbf{v} & \text{if } \mathbf{v} \text{ is } \mathbf{E}\text{'s value} \\ \mathbf{error} & \text{if } \mathbf{E} \text{ causes an error} \end{cases}$$

For example **E**[**1**+**1**]=**2** but **E**[**1**+**true**]=**error**.

To handle (ii) we must make the result of an evaluation a function of the state—i.e. define **E**:**Exp**→[**State**→[**Value**+{**error**}]] so that:

$$\mathbf{E}[\mathbf{E}]\mathbf{s} = \begin{cases} \mathbf{v} & \text{if } \mathbf{v} \text{ is } \mathbf{E}\text{'s value in } \mathbf{s} \\ \mathbf{error} & \text{if the evaluation causes an error} \end{cases}$$

For example:

$$\mathbf{E}[\mathbf{1}+\mathbf{x}]\mathbf{s} = \begin{cases} \mathbf{1}+(\mathbf{mx}) & \text{if } \mathbf{s}=(\mathbf{m},\mathbf{i},\mathbf{o}) \text{ and } \mathbf{mx} \text{ is a number} \\ \mathbf{error} & \text{otherwise} \end{cases}$$

Thus the denotation **E**[**E**] of **E** is a function of type

Exp→[**State**→[**Value**+{**error**}]].

Finally to handle (iii) we must further complicate **E**'s type so that:

E:**Exp**→[**State**→[[**Value**×**State**]+{**error**}]]

and then

$$\mathbf{E}[\mathbf{E}]\mathbf{s} = \begin{cases} (\mathbf{v}, \mathbf{s}') & \text{where } \mathbf{v} \text{ is } \mathbf{E}\text{'s value in } \mathbf{s} \text{ and } \mathbf{s}' \text{ is the state after the evaluation.} \\ \mathbf{error} & \text{if an error occurs.} \end{cases}$$

For example:

$$\mathbf{E[read]s} = \begin{cases} \mathbf{(hd\ i, (m, tl\ i, s))} & \text{if } \mathbf{s = (m, i, o)}, \mathbf{i} \text{ is non empty and has first member } \mathbf{hd\ i} \text{ and the rest is } \mathbf{tl\ i}. \\ \mathbf{error} & \text{if the input is empty.} \end{cases}$$

We formally define **E** by cases on the different kinds of expressions. For example the denotations of expressions of the form **I** are defined by the following *semantic clause*:

$$\mathbf{E[I]\,(m, i, o) = (m\,I = unbound) \rightarrow error, (m\,I, (m, i, o))}$$

Here "$\mathbf{b \rightarrow v_1, v_2}$" means "if **b** is true then $\mathbf{v_1}$ else $\mathbf{v_2}$"—we give a precise account of this notation in 3.4.5. Thus the above semantic clause says that if **m I** is **unbound** (i.e. **I** is unbound in **m**) then an error occurs otherwise the value of **I** is whatever it is bound to in the memory. The state resulting from the evaluation is **(m, i, o)**—i.e. the original state. An example of a semantic clause in which the evaluation changes the state is:

$$\mathbf{E[read]\,(m, i, o) = null\ i \rightarrow error, (hd\ i, (m, tl\ i, o))}$$

Here **null i** is true if **i** is empty, **hd i** is the first element of **i** and **tl i** the rest (see 3.3.3.3.). Thus if the input is empty an error results otherwise the value of **read** is the first item in the input and the resulting state has this item removed.

The rest of the semantic clauses for TINY are described in 2.4.4. below.

2.4.3.2. Denotations of commands

The effect of executing a command is to produce a new state or generate an error, thus:

$$\mathbf{C: Com \rightarrow [State \rightarrow [State + \{error\}]]}$$

a typical semantic clause is:

$$\mathbf{C[output\ E]\ s = (E[E]s = (v, (m, i, o))) \rightarrow (m, i, v\,.\,o), error}$$

Here **v. o** is the string resulting from sticking **v** onto the front of **o**. Thus this semantic clause says that **C[output E]** is a function which when applied to a state **s** first evaluates **E** and if **E** produces a value **v** and new

state **(m, i, o)** then the result is **(m, i, v. o)** otherwise the result is **error**.

2.4.4. Semantic clauses

In this section we describe and explain the semantic clauses for TINY.

2.4.4.1. Clauses for expressions

(E1) $\mathbf{E[0]s = (0, s)}$
$\mathbf{E[1]s = (1, s)}$

The value of a numeral is the corresponding number; the evaluation does not change the state.

(E2) $\mathbf{E[true]s = (true, s)}$
$\mathbf{E[false]s = (false, s)}$

The value of a boolean constant is the corresponding boolean value (truth value); the evaluation does not change the state.

(E3) $\mathbf{E[read]\,(m, i, o) = null\ i \rightarrow error, (hd\ i, (m, tl\ i, o))}$

This was explained above, **null i** is **true** if **i** is empty and **false** otherwise; **hd i** is the first element of **i** and **tl i** is the rest of **i**.

(E4) $\mathbf{E[I]\,(m, i, o) = (m\,I = unbound) \rightarrow error, (m\,I, (m, i, o))}$

This was explained in 2.4.3.1. above.

(E5) $\mathbf{E[not\ E]\ s = (E[E]s = (v, s')) \rightarrow (isBool\ v \rightarrow (not\ v, s'), error), error}$

isBool v is **true** if $\mathbf{v \varepsilon Bool}$ and is **false** otherwise (here $\mathbf{v \varepsilon Value = Num + Bool}$), **not: Bool→Bool** is the function defined by **not true = false** and **not false = true**. Thus the value of **not E** is **not** of the value of **E** (or **error** if **E**'s evaluation leads to a number or **error**) and the state is changed to the state **s'** resulting from **E**'s evaluation in **s**.

(E6) $\mathbf{E[E_1 = E_2]\ s = (E[E_1]s = (v_1, s_1)) \rightarrow ((E[E_2]s_1 = (v_2, s_2)) \rightarrow (v_1 = v_2, s_2), error), error}$

Here $\mathbf{v_1 = v_2}$ is **true** if $\mathbf{v_1}$ equals $\mathbf{v_2}$ and **false** otherwise. Thus the result of $\mathbf{E_1 = E_2}$ in **s** is obtained by first evaluating $\mathbf{E_1}$ in **s** to get $\mathbf{(v_1, s_1)}$ (or **error** —in which case **error** is the value of $\mathbf{E_1 = E_2}$), then evaluating $\mathbf{E_2}$ in $\mathbf{s_1}$ to

get (v_2, s_2) (or **error**—in which case **error** is the value of $E_1 = E_2$), finally $(v_1 = v_2, s_2)$ is returned as the result of $E_1 = E_2$.

(E7) $\mathbf{E}[\![E_1 + E_2]\!]s = (\mathbf{E}[\![E_1]\!]s = (v_1, s_1)) \rightarrow$
$(\mathbf{E}[\![E_2]\!]s_1 = (v_2, s_2)) \rightarrow$
(isNum v_1 **and isNum** $v_2 \rightarrow$
$(v_1 + v_2, s_2)$, **error**), **error**), **error**

This semantic clause is similar to the preceding one. **isNum v** is **true** if **v** is a number (i.e. member of **Num**) and **false** otherwise. Thus (E7) says that to evaluate $E_1 + E_2$ one evaluates E_1 then evaluates E_2 (in the state resulting from E_1) then tests their values to make sure they are numbers and if so returns their sum and the state resulting from E_2. If either of these values is not a number or if E_1 or E_2 generates an error then $E_1 + E_2$ generates an error.

2.4.4.2. Clauses for commands

(C1) $\mathbf{C}[\![I := E]\!]\, s = (\mathbf{E}[\![E]\!]s = (v, (m, c, o))) \rightarrow (m[v/I], i, o)$, **error**

$$(m[v/I])\, I' = \begin{cases} v & \text{if } I = I' \\ m\, I' & \text{otherwise} \end{cases}$$

Thus $\mathbf{C}[\![I := E]\!]\, s$ is a state identical to the state resulting from the evaluation of **E** except that **E**'s value **v** is bound to **I** in the memory (if **E** produces **error** then so does **I := E**).

(C2) $\mathbf{C}[\![$**output E**$]\!]\, s = (\mathbf{E}[\![E]\!]s = (v, (m, i, o))) \rightarrow (m, i, v \,.\, o)$, **error**

The semantic clause was explained in 2.4.3.2. above.

(C3) $\mathbf{C}[\![$**if E then** C_1 **else** $C_2]\!]\, s =$
$((\mathbf{E}[\![E]\!]\, s = (v, s')) \rightarrow$
(isBool $v \rightarrow (v \rightarrow \mathbf{C}[\![C_1]\!]\, s', \mathbf{C}[\![C_2]\!]\, s')$, **error**), **error**)

isBool v is **true** if **v** is a truth value (i.e. if $v \varepsilon$ **Bool** = {**true**, **false**}) and is **false** otherwise. Thus if **E** produces result **(v,s')** when evaluated in **s** then C_1 or C_2 are executed (in the state **s'** resulting from **E**) depending on whether **E**'s value **v** is **true** or **false**.

(C4) $\mathbf{C}[\![$**while E do C**$]\!]\, s =$
$((\mathbf{E}[\![E]\!]\, s = (v, s')) \rightarrow$
(isBool $v \rightarrow$
$(v \rightarrow ((\mathbf{C}[\![C]\!]\, s' = s'') \rightarrow \mathbf{C}[\![$**while E do C**$]\!]\, s'', $**error**$), s'),$
error), **error**)

If **E** produces an error then so does **while E do C**; if **E** produces value **v** and a new state **s'** then if **v** is **true** **C** is done to get **s"** and then **while E do C** is done starting with **s"**. If **v** is **false** then the result of **while E do C** is **s'** the result of **E**. Finally if **v** is not a truth value or $\mathbf{C}[\![C]\!]s' =$ **error** then **while E do C** causes an error. Notice that (C4) is *recursive* $\mathbf{C}[\![$**while E do s**$]\!]$ occurs on the right hand side.

(C5) $\mathbf{C}[\![C_1; C_2]\!]\, s = (\mathbf{C}[\![C_1]\!]s = $ **error**$) \rightarrow$ **error**$, \mathbf{C}[\![C_2]\!](\mathbf{C}[\![C_1]\!]s)$

Thus if C_1 generates an error then so does $C_1; C_2$ otherwise C_2 is done in the state resulting from C_1.

This completes the semantic clauses, and also the formal semantics, of TINY. In some of the clauses, especially (E6). (E7), (C3) and (C4), the tests for the various error conditions makes it hard to follow what is going on—the semantics of non error executions fails to stand out. In the next chapter we describe various notations for clarifying and simplifying semantic clauses; the reader may like to peep ahead to chapter 4 to see how neat the clauses can be made.

An important thing to note about the semantic clauses above is that each construct has its denotation defined in terms of the denotation of its components; for example $\mathbf{C}[\![C_1; C_2]\!]$ is defined in terms of $\mathbf{C}[\![C_1]\!]$ and $\mathbf{C}[\![C_2]\!]$. The only exception to this is (C4), the clause for **while E do C**, where the denotation is defined 'in terms of itself'—we shall have more to say on such recursive definitions in 3.2.1.

2.4.5. Summary of the formal semantics of TINY

The formal semantic description of TINY just given had three main parts:

(i) Specification of the syntactic domains **Exp** and **Com**.
(ii) Specification of the semantic domains **State**, **Value** etc.
(iii) Specification of the semantic functions $\mathbf{E}$ and $\mathbf{C}$ which map syntactic entities to semantic ones.

In the next chapter we shall describe in detail the concepts and notations of these specifications. In subsequent chapters we shall refine and extend the concepts and notation, and apply the techniques to a wide variety of programming constructs (including most of ALGOL 60 and PASCAL).