SEMANTICS WITH APPLICATIONS
A Formal Introduction

©*Hanne Riis Nielson* ©*Flemming Nielson*

This is a revised edition completed in July 1999; the original edition from 1992 was published by John Wiley & Sons; this should be acknowledged in all references to the book.

Contents

List of Tables

Preface

Many books on formal semantics begin by explaining that there are three major approaches to semantics, that is

- operational semantics,
- denotational semantics, and
- axiomatic semantics;

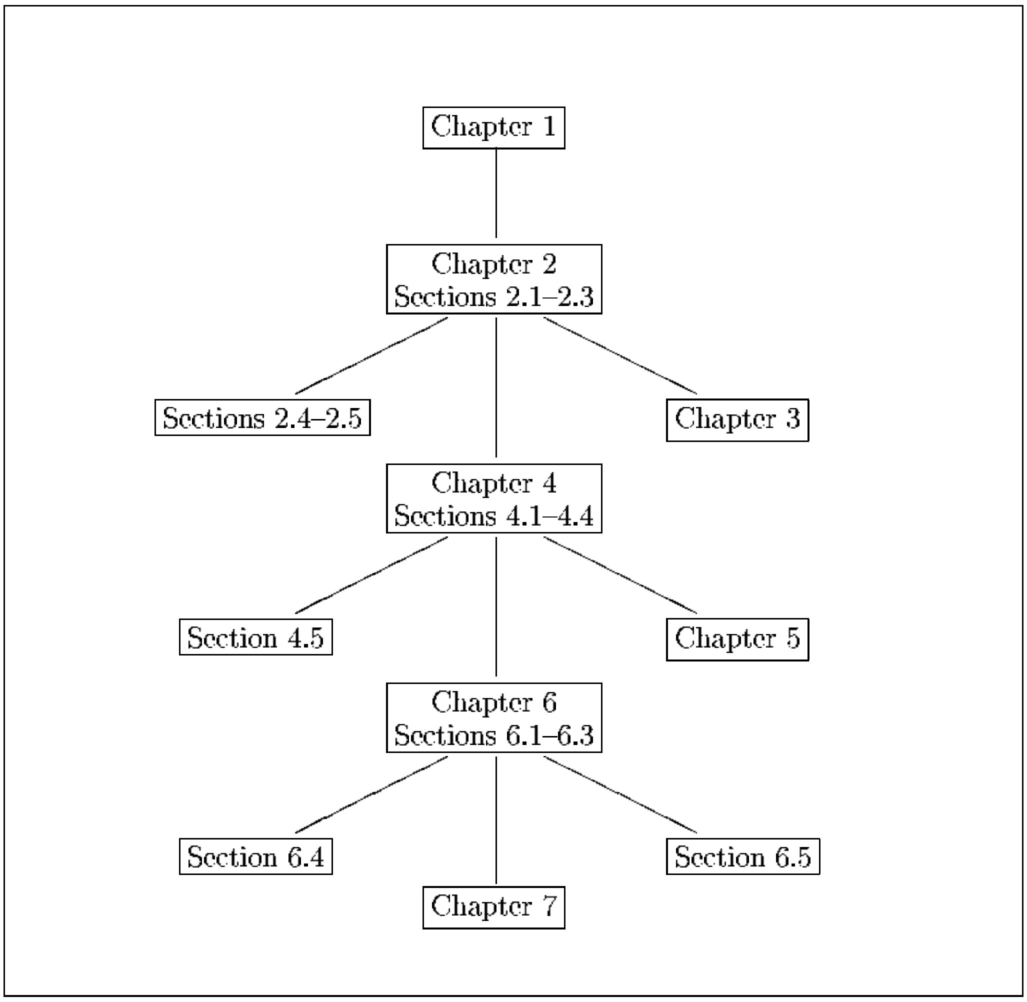
but then they go on to study just *one* of these in greater detail. The purpose of this book is to

- present the *fundamental ideas* behind *all* of these approaches,
- to stress their *relationship* by formulating and proving the relevant theorems, and
- to illustrate the *applicability* of formal semantics as a tool in computer science.

This is an ambitious goal and to achieve it, the bulk of the development concentrates on a rather small core language of `while`-programs for which the three approaches are developed to roughly the same level of sophistication. To demonstrate the *applicability* of formal semantics we show

- how to use semantics for validating prototype implementations of programming languages,
- how to use semantics for verifying analyses used in more advanced implementations of programming languages, and
- how to use semantics for verifying useful program properties including information about execution time.

The development is *introductory* as is already reflected in the title. For this reason very many advanced concepts within operational, denotational and axiomatic semantics have had to be omitted. Also we have had to omit treatment of other approaches to semantics, for example Petri-nets and temporal logic. Some pointers to further reading are given in Chapter 7.



Overview

As is illustrated in the dependency diagram, Chapters 1, 2, 4, 6 and 7 form the core of the book. Chapter 1 introduces the example language of `while`-programs that is used throughout the book. In Chapter 2 we cover two approaches to *operational semantics*, the natural semantics of G. Kahn and the structural operational semantics of G. Plotkin. Chapter 4 develops the *denotational semantics* of D. Scott and C. Strachey including simple fixed point theory. Chapter 6 introduces *program verification* based on operational and denotational semantics and goes on to present the axiomatic approach due to C. A. R. Hoare. Finally, Chapter 7 contains suggestions for further reading.

The first three or four sections of each of the Chapters 2, 4 and 6 are devoted to the language of `while`-programs and covers specification as well as theoretical aspects. In each of the chapters we extend the `while`-language with various other constructs and the emphasis is here on specification rather than theory. In Sections 2.4 and 2.5 we consider extensions with abortion, non-determinism, parallelism, block constructs, dynamic and static procedures, and non-recursive and recursive procedures. In Section 4.5 we consider extensions of the `while`-language with static procedures that may or may not be recursive and we show how to handle exceptions, that is, certain kinds of jumps. Finally, in Section 6.4 we consider an extension with non-recursive and recursive procedures and we also show how total correctness properties are handled. The sections on extending the operational, denotational and axiomatic semantics may be studied in any order.

The applicability of operational, denotational and axiomatic semantics is illustrated in Chapters 3, 5 and 6. In Chapter 3 we show how to prove the correctness of a simple compiler for the `while`-language using the operational semantics. In Chapter 5 we prove an analysis for the `while`-language correct using the denotational semantics. Finally, in Section 6.5 we extend the axiomatic approach so as to obtain information about execution time of `while`-programs.

Appendix A reviews the mathematical notation on which this book is based. It is mostly standard notation but some may find our use of $\hookrightarrow$ and $\diamond$ non-standard. We use $D \hookrightarrow E$ for the set of *partial* functions from D to E; this is because we find that the $D \rightharpoonup E$ notation is too easily overlooked. Also we use $R \diamond S$ for the composition of binary relations R and S; this is because of the different order of composition used for relations and functions. When dealing with axiomatic semantics we use formulae $\{\ P\ \}\ S\ \{\ Q\ \}$ for partial correctness assertions but $\{\ P\ \}\ S\ \{\ \Downarrow Q\ \}$ for total correctness assertions because the explicit occurrence of $\Downarrow$ (for termination) may prevent the student from confusing the two systems.

Appendices B, C and D contain implementations of some of the semantic specifications using the functional language **Miranda.**[1] The intention is that the ability to experiment with semantic definitions enhances the understanding of material that is often regarded as being terse and heavy with formalism. It should be possible to rework these implementations in any functional language but if an eager language (like **Standard ML**) is used, great care must be taken in the implementation of the fixed point combinator. However, no continuity is lost if these appendices are ignored.

Notes for the instructor

The reader should preferably be acquainted with the BNF-style of specifying the syntax of programming languages and should be familiar with most of the mathematical concepts surveyed in Appendix A. To appreciate the prototype implementations of the appendices some experience in functional programming is required.

[1]**Miranda** is a trademark of Research Software Limited, 23 St Augustines Road, Canterbury, Kent CT1 1XP, UK.

We have ourselves used this book for an undergraduate course at Aarhus University in which the required functional programming is introduced "on-the-fly".

We provide two kinds of exercises. One kind helps the student in his/her understanding of the definitions/results/techniques used in the text. In particular there are exercises that ask the student to prove auxiliary results needed for the main results but then the proof techniques will be minor variations of those already explained in the text. We have marked those exercises whose results are needed later by "**(Essential)**". The other kind of exercises are more challenging in that they extend the development, for example by relating it to other approaches. We use a star to mark the more difficult of these exercises. Exercises marked by two stars are rather lengthy and may require insight not otherwise presented in the book. It will not be necessary for students to attempt all the exercises but we do recommend that they read them and try to understand what the exercises are about.

Acknowledgements

In writing this book we have been greatly assisted by the comments and suggestions provided by colleagues and reviewers and by students and instructors at Aarhus University. This includes Anders Gammelgaard, Chris Hankin, Torben Amtoft Hansen, Jens Palsberg Jørgensen, Ernst-Rüdiger Olderog, David A. Schmidt, Kirsten L. Solberg and Bernhard Steffen. Special thanks are due to Steffen Grarup, Jacob Seligmann, and Bettina Blaaberg Sørensen for their enthusiasm and great care in reading preliminary versions.

Aarhus, October 1991 Hanne Riis Nielson

Flemming Nielson

Revised Edition

In this revised edition we have corrected a number of typographical errors and a few mistakes; however, no major changes have been made. Since the publication of the first edition we have obtained helpful comments from Jens Knoop and Anders Sandholm. The webpage for the book now also contains implementations of Appendices B, C and D in Gofer as well as in Miranda.

Aarhus, July 1999 Hanne Riis Nielson

Flemming Nielson

Chapter 1

Introduction

The purpose of this book is

- to describe some of the main ideas and methods used in semantics,
- to illustrate these on interesting applications, and
- to investigate the relationship between the various methods.

Formal semantics is concerned with rigorously specifying the meaning, or behaviour, of programs, pieces of hardware etc. The need for rigour arises because

- it can reveal ambiguities and subtle complexities in apparently crystal clear defining documents (for example programming language manuals), and
- it can form the basis for implementation, analysis and verification (in particular proofs of correctness).

We will use informal set theoretic notation (reviewed in Appendix A) to represent semantic concepts. This will suffice in this book but for other purposes greater notational precision (that is, formality) may be needed, for example when processing semantic descriptions by machine as in semantics directed compiler-compilers or machine assisted proof checkers.

1.1 Semantic description methods

It is customary to distinguish between the syntax and the semantics of a programming language. The *syntax* is concerned with the grammatical structure of programs. So a syntactic analysis of the program

```
z:=x; x:=y; y:=z
```

will realize that it consists of three statements separated by the symbol ';'. Each of these statements has the form of a variable followed by the composite symbol ':=' and an expression which is just a variable.

The *semantics* is concerned with the meaning of grammatically correct programs. So it will express that the meaning of the above program is to exchange the values of the variables `x` and `y` (and setting `z` to the final value of `y`). If we were to explain this in more detail we would look at the grammatical structure of the program and use explanations of the meanings of

- sequences of statements separated by ';', and
- a statement consisting of a variable followed by ':=' and an expression.

The actual explanations can be formalized in different ways. In this book we shall consider three approaches. Very roughly, the ideas are as follows:

Operational semantics: The meaning of a construct is specified by the computation it induces when it is executed on a machine. In particular, it is of interest *how* the effect of a computation is produced.

Denotational semantics: Meanings are modelled by mathematical objects that represent the effect of executing the constructs. Thus *only* the effect is of interest, not how it is obtained.

Axiomatic semantics: Specific properties of the effect of executing the constructs are expressed as *assertions*. Thus there may be aspects of the executions that are ignored.

To get a feeling for their different nature let us see how they express the meaning of the example program above.

Operational semantics (Chapter 2)

An operational explanation of the meaning of a construct will tell how to *execute* it:

- To execute a sequence of statements separated by ';' we execute the individual statements one after the other and from left to right.
- To execute a statement consisting of a variable followed by ':=' and another variable we determine the value of the second variable and assign it to the first variable.

We shall record the execution of the example program in a state where `x` has the value **5**, `y` the value **7** and `z` the value **0** by the following "derivation sequence":

$$\begin{array}{lrl}
 & \langle \texttt{z:=x; x:=y; y:=z}, & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}]\rangle \\
\Rightarrow & \langle \texttt{x:=y; y:=z}, & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}]\rangle \\
\Rightarrow & \langle \texttt{y:=z}, & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}]\rangle \\
\Rightarrow & & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{5}, \texttt{z}\mapsto\mathbf{5}]
\end{array}$$

In the first step we execute the statement z:=x and the value of z is changed to **5** whereas those of x and y are unchanged. The remaining program is now x:=y; y:=z. After the second step the value of x is **7** and we are left with the program y:=z. The third and final step of the computation will change the value of y to **5**. Therefore the initial values of x and y have been exchanged, using z as a temporary variable.

This explanation gives an *abstraction* of how the program is executed on a machine. It is important to observe that it is indeed an abstraction: we ignore details like use of registers and addresses for variables. So the operational semantics is rather independent of machine architectures and implementation strategies.

In Chapter 2 we shall formalize this kind of operational semantics which is often called *structural operational semantics* (or small-step semantics). An alternative operational semantics is called *natural semantics* (or big-step semantics) and differs from the structural operational semantics by hiding even more execution details. In the natural semantics the execution of the example program in the same state as before will be represented by the following "derivation tree":

$$\frac{\dfrac{\langle \texttt{z:=x}, s_0\rangle \to s_1 \qquad \langle \texttt{x:=y}, s_1\rangle \to s_2}{\langle \texttt{z:=x; x:=y}, s_0\rangle \to s_2} \qquad \langle \texttt{y:=z}, s_2\rangle \to s_3}{\langle \texttt{z:=x; x:=y; y:=z}, s_0\rangle \to s_3}$$

where we have used the abbreviations:

$$\begin{array}{lcl}
s_0 & = & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}] \\
s_1 & = & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}] \\
s_2 & = & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}] \\
s_3 & = & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{5}, \texttt{z}\mapsto\mathbf{5}]
\end{array}$$

This is to be read as follows: The execution of z:=x in the state s_0 will result in the state s_1 and the execution of x:=y in state s_1 will result in state s_2. Therefore the execution of z:=x; x:=y in state s_0 will give state s_2. Furthermore, execution of y:=z in state s_2 will give state s_3 so in total the execution of the program in state s_0 will give the resulting state s_3. This is expressed by

$$\langle \texttt{z:=x; x:=y; y:=z}, s_0\rangle \to s_3$$

but now we have hidden the above explanation of how it was actually obtained.

In Chapter 3 we shall use the natural semantics as the basis for proving the correctness of an implementation of a simple programming language.

Denotational semantics (Chapter 4)

In the denotational semantics we concentrate on the *effect* of executing the programs and we shall model this by mathematical functions:

- The effect of a sequence of statements separated by ';' is the functional composition of the effects of the individual statements.
- The effect of a statement consisting of a variable followed by ':=' and another variable is the function that given a state will produce a new state: it is as the original one except that the value of the first variable of the statement is equal to that of the second variable.

For the example program we obtain functions written $\mathcal{S}[\![\texttt{z:=x}]\!]$, $\mathcal{S}[\![\texttt{x:=y}]\!]$, and $\mathcal{S}[\![\texttt{y:=z}]\!]$ for each of the assignment statements and for the overall program we get the function

$$\mathcal{S}[\![\texttt{z:=x; x:=y; y:=z}]\!] = \mathcal{S}[\![\texttt{y:=z}]\!] \circ \mathcal{S}[\![\texttt{x:=y}]\!] \circ \mathcal{S}[\![\texttt{z:=x}]\!]$$

Note that the *order* of the statements have changed because we use the usual notation for function composition where $(f \circ g)\ s$ means $f\ (g\ s)$. If we want to determine the effect of executing the program on a particular state then we can *apply* the function to that state and *calculate* the resulting state as follows:

$$\begin{aligned}
&\mathcal{S}[\![\texttt{z:=x; x:=y; y:=z}]\!]([\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}])\\
&\quad= (\mathcal{S}[\![\texttt{y:=z}]\!] \circ \mathcal{S}[\![\texttt{x:=y}]\!] \circ \mathcal{S}[\![\texttt{z:=x}]\!])([\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}])\\
&\quad= \mathcal{S}[\![\texttt{y:=z}]\!](\mathcal{S}[\![\texttt{x:=y}]\!](\mathcal{S}[\![\texttt{z:=x}]\!]([\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}])))\\
&\quad= \mathcal{S}[\![\texttt{y:=z}]\!](\mathcal{S}[\![\texttt{x:=y}]\!]([\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}]))\\
&\quad= \mathcal{S}[\![\texttt{y:=z}]\!]([\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}])\\
&\quad= [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{5}, \texttt{z}\mapsto\mathbf{5}]
\end{aligned}$$

Note that we are only manipulating mathematical objects; we are not concerned with executing programs. The difference may seem small for a program with only assignment and sequencing statements but for programs with more sophisticated constructs it is substantial. The benefits of the denotational approach are mainly due to the fact that it abstracts away from how programs are executed. Therefore it becomes easier to reason about programs as it simply amounts to reasoning about mathematical objects. However, a prerequisite for doing so is to establish a firm mathematical basis for denotational semantics and this task turns out not to be entirely trivial.

The denotational approach can easily be adapted to express other sorts of properties of programs. Some examples are:

- Determine whether all variables are initialized before they are used — if not a warning may be appropriate.
- Determine whether a certain expression in the program always evaluates to a constant — if so one can replace the expression by the constant.
- Determine whether all parts of the program are reachable — if not they could as well be removed or a warning might be appropriate.

In Chapter 5 we develop an example of this.

While we prefer the denotational approach when reasoning about programs we may prefer an operational approach when implementing the language. It is therefore of interest whether a denotational definition is *equivalent* to an operational definition and this is studied in Section 4.3.

Axiomatic semantics (Chapter 6)

Often one is interested in *partial correctness properties* of programs: A program is partially correct, with respect to a precondition and a postcondition, if whenever the initial state fulfils the precondition and the program terminates, then the final state is guaranteed to fulfil the postcondition. For our example program we have the partial correctness property:

$$\{\ \texttt{x}=\texttt{n} \land \texttt{y}=\texttt{m}\ \}\ \texttt{z:=x; x:=y; y:=z}\ \{\ \texttt{y}=\texttt{n} \land \texttt{x}=\texttt{m}\ \}$$

where x=n ∧ y=m is the precondition and y=n ∧ x=m is the postcondition. The names n and m are used to "remember" the initial values of x and y, respectively. The state [x↦**5**, y↦**7**, z↦**0**] satisfies the precondition by taking n=**5** and m=**7** and when we have *proved* the partial correctness property we can deduce that *if* the program terminates *then* it will do so in a state where y is **5** and x is **7**. However, the partial correctness property does not ensure that the program *will* terminate although this is clearly the case for the example program.

The axiomatic semantics provides a *logical system* for proving partial correctness properties of individual programs. A proof of the above partial correctness property may be expressed by the following "proof tree":

$$\frac{\dfrac{\{\ p_0\ \}\ \mathtt{z{:=}x}\ \{\ p_1\ \} \qquad \{\ p_1\ \}\ \mathtt{x{:=}y}\ \{\ p_2\ \}}{\{\ p_0\ \}\ \mathtt{z{:=}x;\ x{:=}y}\ \{\ p_2\ \}} \qquad \{\ p_2\ \}\ \mathtt{y{:=}z}\ \{\ p_3\ \}}{\{\ p_0\ \}\ \mathtt{z{:=}x;\ x{:=}y;\ y{:=}z}\ \{\ p_3\ \}}$$

where we have used the abbreviations

$$\begin{aligned} p_0 &= \mathtt{x{=}n} \land \mathtt{y{=}m} \\ p_1 &= \mathtt{z{=}n} \land \mathtt{y{=}m} \\ p_2 &= \mathtt{z{=}n} \land \mathtt{x{=}m} \\ p_3 &= \mathtt{y{=}n} \land \mathtt{x{=}m} \end{aligned}$$

We may view the logical system as a specification of only certain aspects of the semantics. It usually does not capture all aspects for the simple reason that all the partial correctness properties listed below can be proved using the logical system but certainly we would not regard the programs as behaving in the same way:

$\{\ \mathtt{x{=}n} \land \mathtt{y{=}m}\ \}\ \mathtt{z{:=}x;\ x{:=}y;\ y{:=}z}\ \{\ \mathtt{y{=}n} \land \mathtt{x{=}m}\ \}$

$\{\ \mathtt{x{=}n} \land \mathtt{y{=}m}\ \}\ \mathtt{if\ x{=}y\ then\ skip\ else}\ (\mathtt{z{:=}x;\ x{:=}y;\ y{:=}z})\ \{\ \mathtt{y{=}n} \land \mathtt{x{=}m}\ \}$

$\{\ \mathtt{x{=}n} \land \mathtt{y{=}m}\ \}\ \mathtt{while\ true\ do\ skip}\ \{\ \mathtt{y{=}n} \land \mathtt{x{=}m}\ \}$

The benefits of the axiomatic approach are that the logical systems provide an easy way of proving properties of programs — and to a large extent it has been possible to automate it. Of course this is only worthwhile if the axiomatic semantics is faithful to the "more general" (denotational or operational) semantics we have in mind and we shall discuss this in Section 6.3.

The complementary view

It is important to note that these kinds of semantics are *not* rival approaches, but are different techniques appropriate for different purposes and — to some extent — for different programming languages. To stress this, the development will address the following issues:

- It will develop each of the approaches for a simple language of `while`-programs.
- It will illustrate the power and weakness of each of the approaches by extending the `while`-language with other programming constructs.
- It will prove the relationship between the approaches for the `while`-language.

- It will give examples of applications of the semantic descriptions in order to illustrate their merits.

1.2 The example language While

This book illustrates the various forms of semantics on a very simple imperative programming language called **While**. As a first step we must specify its syntax.

The syntactic notation we use is based on BNF. First we list the various *syntactic categories* and give a meta-variable that will be used to range over *constructs* of each category. For our language the meta-variables and categories are as follows:

n will range over numerals, **Num**,

x will range over variables, **Var**,

a will range over arithmetic expressions, **Aexp**,

b will range over boolean expressions, **Bexp**, and

S will range over statements, **Stm**.

The meta-variables can be primed or subscripted. So, for example, n, n', n_1, n_2 all stand for numerals.

We assume that the structure of numerals and variables is given elsewhere; for example numerals might be strings of digits, and variables strings of letters and digits starting with a letter. The structure of the other constructs is:

$$\begin{array}{lcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \texttt{skip} \mid S_1 \,;\, S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ S
\end{array}$$

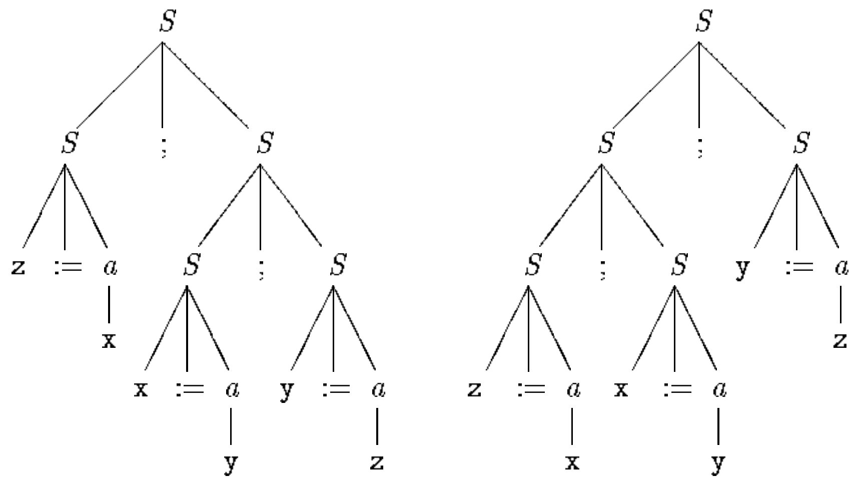
Thus, a boolean expression b can only have one of six forms. It is called a *basis element* if it is `true` or `false` or has the form $a_1 = a_2$ or $a_1 \leq a_2$ where a_1 and a_2 are arithmetic expressions. It is called a *composite element* if it has the form $\neg b$ where b is a boolean expression, or the form $b_1 \wedge b_2$ where b_1 and b_2 are boolean expressions. Similar remarks apply to arithmetic expressions and statements.

The specification above defines the *abstract syntax* of **While** in that it simply says how to build arithmetic expressions, boolean expressions and statements in the language. One way to think of the abstract syntax is as specifying the parse trees of the language and it will then be the purpose of the *concrete syntax* to provide sufficient information that enable unique parse trees to be constructed.

So given the string of characters:

z:=x; x:=y; y:=z

the concrete syntax of the language must be able to resolve which of the two abstract syntax trees below it is intended to represent:



In this book we shall *not* be concerned with concrete syntax. Whenever we talk about syntactic entities such as arithmetic expressions, boolean expressions or statements we will always be talking about the abstract syntax so there is no ambiguity with respect to the form of the entity. In particular, the two trees above are both elements of the syntactic category **Stm**.

It is rather cumbersome to use the graphical representation of abstract syntax and we shall therefore use a linear notation. So we shall write

z:=x; (x:=y; y:=z)

for the leftmost syntax tree and

(z:=x; x:=y); y:=z

for the rightmost one. For statements one often writes the brackets as `begin` ⋯ `end` but we shall feel free to use (⋯) in this book. Similarly, we use brackets (⋯) to resolve ambiguities for elements in the other syntactic categories. To cut down on the number of brackets needed we shall allow to use the familiar relative binding powers (precedences) of $+$, $\star$ and $-$ etc. and so write 1+x⋆2 for 1+(x⋆2) but not for (1+x)⋆2.

Exercise 1.1 The following statement is in **While**:

y:=1; `while` ¬(x=1) `do` (y:=y⋆x; x:=x−1)

It computes the factorial of the initial value bound to x (provided that it is positive) and the result will be the final value of y. Draw a graphical representation of the abstract syntax tree. □

Exercise 1.2 Assume that the initial value of the variable x is n and that the initial value of y is m. Write a statement in **While** that assigns z the value of n to the power of m, that is

$$\underbrace{n \star \cdots \star n}_{m \text{ times}}$$

Give a linear as well as a graphical representation of the abstract syntax. □

The semantics of **While** is given by defining so-called *semantic functions* for each of the syntactic categories. The idea is that a semantic function takes a syntactic entity as argument and returns its meaning. The operational, denotational and axiomatic approaches mentioned earlier will be used to specify semantic functions for the statements of **While**. For numerals, arithmetic expressions and boolean expressions the semantic functions are specified once and for all below.

1.3 Semantics of expressions

Before embarking on specifying the semantics of the arithmetic and boolean expressions of **While** let us have a brief look at the numerals; this will present the main ingredients of the approach in a very simple setting. So assume for the moment that the numerals are in the *binary* system. Their abstract syntax could then be specified by:

$$n ::= \mathsf{0} \mid \mathsf{1} \mid n\ \mathsf{0} \mid n\ \mathsf{1}$$

In order to determine the number represented by a numeral we shall define a function

$$\mathcal{N}\colon \mathbf{Num} \to \mathbf{Z}$$

This is called a *semantic function* as it defines the semantics of the numerals. We want $\mathcal{N}$ to be a *total function* because we want to determine a unique number for each numeral of **Num**. If $n \in \mathbf{Num}$ then we write $\mathcal{N}[\![n]\!]$ for the application of $\mathcal{N}$ to n, that is for the corresponding number. In general, the application of a semantic function to a syntactic entity will be written within the "syntactic" brackets '$[\![$' and '$]\!]$' rather than the more usual '(' and ')'. These brackets have no special meaning but throughout this book we shall enclose syntactic arguments to semantic functions using the "syntactic" brackets whereas we use ordinary brackets (or juxtapositioning) in all other cases.

The semantic function $\mathcal{N}$ is defined by the following *semantic clauses* (or *equations*):

$$\begin{aligned}
\mathcal{N}[\![0]\!] &= \mathbf{0}\\
\mathcal{N}[\![1]\!] &= \mathbf{1}\\
\mathcal{N}[\![n\ 0]\!] &= \mathbf{2} \star \mathcal{N}[\![n]\!]\\
\mathcal{N}[\![n\ 1]\!] &= \mathbf{2} \star \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}$$

Here $\mathbf{0}$ and $\mathbf{1}$ are numbers, that is elements of $\mathbf{Z}$. Furthermore, $\star$ and $+$ are the usual arithmetic operations on numbers. The above definition is an example of a *compositional* definition; this means that for each possible way of constructing a numeral it tells how the corresponding number is obtained from the meanings of the *sub*constructs.

Example 1.3 We can calculate the number $\mathcal{N}[\![101]\!]$ corresponding to the numeral 101 as follows:

$$\begin{aligned}
\mathcal{N}[\![101]\!] &= \mathbf{2} \star \mathcal{N}[\![10]\!] + \mathbf{1}\\
&= \mathbf{2} \star (\mathbf{2} \star \mathcal{N}[\![1]\!]) + \mathbf{1}\\
&= \mathbf{2} \star (\mathbf{2} \star \mathbf{1}) + \mathbf{1}\\
&= \mathbf{5}
\end{aligned}$$

Note that the string 101 is decomposed according to the syntax for numerals. □

So far we have only *claimed* that the definition of $\mathcal{N}$ gives rise to a well-defined total function. We shall now present a *formal proof* showing that this is indeed the case.

Fact 1.4 The above equations for $\mathcal{N}$, define a total function $\mathcal{N}$: $\mathbf{Num} \to \mathbf{Z}$.

Proof: We have a total function $\mathcal{N}$, if for all arguments $n \in \mathbf{Num}$

there is exactly one number $\mathbf{n} \in \mathbf{Z}$ such that $\mathcal{N}[\![n]\!] = \mathbf{n}$ (*)

Given a numeral n it can have one of four forms: it can be a basis element and then it is equal to 0 or 1, or it can be a composite element and then it is equal to $n'0$ or $n'1$ for some other numeral n'. So, in order to prove (*) we have to consider all four possibilities.

The proof will be conducted by *induction* on the *structure* of the numeral n. In the *base case* we prove (*) for the basis elements of **Num**, that is for the cases where n is 0 or 1. In the *induction step* we consider the composite elements of **Num**, that is the cases where n is $n'0$ or $n'1$. The induction hypothesis will then allow us to assume that (*) holds for the immediate constituent of n, that is n'. We shall then prove that (*) holds for n. It then follows that (*) holds for all numerals n because any numeral n can be constructed in that way.

The case $n = 0$: Only one of the semantic clauses defining $\mathcal{N}$ can be used and it gives $\mathcal{N}[\![n]\!] = \mathbf{0}$. So clearly there is exactly one number $\mathbf{n}$ in $\mathbf{Z}$ (namely $\mathbf{0}$) such that $\mathcal{N}[\![n]\!] = \mathbf{n}$.

The case $n = 1$ is similar and we omit the details.

The case $n = n'0$: Inspection of the clauses defining $\mathcal{N}$ shows that only one of the clauses is applicable and we have $\mathcal{N}[\![n]\!] = \mathbf{2} \star \mathcal{N}[\![n']\!]$. We can now apply the induction hypothesis to n' and get that there is exactly one number $\mathbf{n}'$ such that $\mathcal{N}[\![n']\!] = \mathbf{n}'$. But then it is clear that there is exactly one number $\mathbf{n}$ (namely $\mathbf{2} \star \mathbf{n}'$) such that $\mathcal{N}[\![n]\!] = \mathbf{n}$.

The case $n = n'1$ is similar and we omit the details. $\square$

The general technique that we have applied in the definition of the syntax and semantics of numerals can be summarized as follows:

Compositional Definitions

1: The syntactic category is specified by an abstract syntax giving the *basis elements* and the *composite elements*. The composite elements have a unique decomposition into their immediate constituents.

2: The semantics is defined by *compositional* definitions of a function: There is a *semantic clause* for each of the basis elements of the syntactic category and one for each of the methods for constructing composite elements. The clauses for composite elements are defined in terms of the semantics of the immediate constituents of the elements.

The proof technique we have applied is closely connected with the approach to defining semantic functions. It can be summarized as follows:

Structural Induction

1: Prove that the property holds for all the *basis* elements of the syntactic category.

2: Prove that the property holds for all the *composite* elements of the syntactic category: Assume that the property holds for all the immediate constituents of the element (this is called the *induction hypothesis*) and prove that it also holds for the element itself.

In the remainder of this book we shall assume that numerals are in decimal notation and have their normal meanings (so for example $\mathcal{N}[\![137]\!] = \mathbf{137} \in \mathbf{Z}$). It is important to understand, however, that there is a distinction between numerals (which are syntactic) and numbers (which are semantic), even in decimal notation.

Semantic functions

The meaning of an expression depends on the values bound to the variables that occur in it. For example, if x is bound to **3** then the arithmetic expression x+1 evaluates to **4** but if x is bound to **2** then the expression evaluates to **3**. We shall therefore introduce the concept of a *state*: to each variable the state will associate its current value. We shall represent a state as a function from variables to values, that is an element of the set

$$\mathbf{State} = \mathbf{Var} \to \mathbf{Z}$$

Each state s specifies a value, written $s\ x$, for each variable x of **Var**. Thus if $s\ \mathtt{x} = \mathbf{3}$ then the value of x+1 in state s is **4**.

Actually, this is just one of several representations of the state. Some other possibilities are to use a table:

x	**5**
y	**7**
z	**0**

or a "list" of the form

$$[\mathtt{x}\mapsto\mathbf{5},\ \mathtt{y}\mapsto\mathbf{7},\ \mathtt{z}\mapsto\mathbf{0}]$$

(as in Section 1.1). In all cases we must ensure that exactly one value is associated with each variable. By requiring a state to be a function this is trivially fulfilled whereas for the alternative representations above extra restrictions have to be enforced.

Given an arithmetic expression a and a state s we can determine the value of the expression. Therefore we shall define the meaning of arithmetic expressions as a total function $\mathcal{A}$ that takes two arguments: the syntactic construct *and* the state. The functionality of $\mathcal{A}$ is

$$\mathcal{A}\text{: } \mathbf{Aexp} \to (\mathbf{State} \to \mathbf{Z})$$

This means that $\mathcal{A}$ takes its parameters *one at a time*. So we may supply $\mathcal{A}$ with its first parameter, say x+1, and study the function $\mathcal{A}[\![\mathtt{x+1}]\!]$. It has functionality **State** → **Z** and only when we supply it with a state (which happens to be a function but that does not matter) do we obtain the value of the expression x+1.

Assuming the existence of the function $\mathcal{N}$ defining the meaning of numerals, we can define the function $\mathcal{A}$ by defining its value $\mathcal{A}[\![a]\!]s$ on each arithmetic expression

$$\begin{array}{lcl}
\mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s & = & s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \star a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \star \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{array}$$

Table 1.1: The semantics of arithmetic expressions

a and state s. The definition of $\mathcal{A}$ is given in Table 1.1. The clause for n reflects that the value of n in any state is $\mathcal{N}[\![n]\!]$. The value of a variable x in state s is the value bound to x in s, that is $s\ x$. The value of the composite expression a_1+a_2 in s is the sum of the values of a_1 and a_2 in s. Similarly, the value of $a_1 \star a_2$ in s is the product of the values of a_1 and a_2 in s, and the value of $a_1 - a_2$ in s is the difference between the values of a_1 and a_2 in s. Note that $+$, $\star$ and $-$ occurring on the right of these equations are the usual arithmetic operations, whilst on the left they are just pieces of syntax; this is analogous to the distinction between numerals and numbers but we shall not bother to use different symbols.

Example 1.5 Suppose that $s\ \mathtt{x} = \mathbf{3}$. Then:

$$\begin{array}{lcl}
\mathcal{A}[\![\mathtt{x}+1]\!]s & = & \mathcal{A}[\![\mathtt{x}]\!]s + \mathcal{A}[\![1]\!]s \\
 & = & (s\ \mathtt{x}) + \mathcal{N}[\![1]\!] \\
 & = & \mathbf{3} + \mathbf{1} \\
 & = & \mathbf{4}
\end{array}$$

Note that here 1 is a numeral (enclosed in the brackets '$[\![$' and '$]\!]$') whereas **1** is a number. □

Example 1.6 Suppose we add the arithmetic expression $-\ a$ to our language. An acceptable semantic clause for this construct would be

$$\mathcal{A}[\![-\ a]\!]s = \mathbf{0} - \mathcal{A}[\![a]\!]s$$

whereas the alternative clause $\mathcal{A}[\![-\ a]\!]s = \mathcal{A}[\![0\ -\ a]\!]s$ would contradict the compositionality requirement. □

Exercise 1.7 Prove that the equations of Table 1.1 define a total function $\mathcal{A}$ in **Aexp** $\to$ (**State** $\to$ **Z**): First argue that it is sufficient to prove that for each $a \in$ **Aexp** and each $s \in$ **State** there is exactly one value $\mathbf{v} \in \mathbf{Z}$ such that $\mathcal{A}[\![a]\!]s = \mathbf{v}$. Next use structural induction on the arithmetic expressions to prove that this is indeed the case. □

$$
\begin{array}{lcl}
\mathcal{B}[\![\texttt{true}]\!]s & = & \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s & = & \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg\, b]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}
\end{array}
$$

Table 1.2: The semantics of boolean expressions

The values of boolean expressions are truth values so in a similar way we shall define their meanings by a (total) function from **State** to **T**:

$$\mathcal{B}\colon \mathbf{Bexp} \to (\mathbf{State} \to \mathbf{T})$$

Here **T** consists of the truth values **tt** (for true) and **ff** (for false).

Using $\mathcal{A}$ we can define $\mathcal{B}$ by the semantic clauses of Table 1.2. Again we have the distinction between syntax (e.g. $\leq$ on the left-hand side) and semantics (e.g. $\leq$ on the right-hand side).

Exercise 1.8 Assume that $s\ \texttt{x} = \mathbf{3}$ and determine $\mathcal{B}[\![\neg(\texttt{x} = \texttt{1})]\!]s$. □

Exercise 1.9 Prove that the equations of Table 1.2 define a total function $\mathcal{B}$ in **Bexp** → (**State** → **T**). □

Exercise 1.10 The syntactic category **Bexp′** is defined as the following extension of **Bexp**:

$$
\begin{array}{lcl}
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 \\
 & \mid & a_1 < a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
 & \mid & b_1 \Rightarrow b_2 \mid b_1 \Leftrightarrow b_2
\end{array}
$$

Give a *compositional* extension of the semantic function $\mathcal{B}$ of Table 1.2.

Two boolean expressions b_1 and b_2 are *equivalent* if for all states s,

$$\mathcal{B}[\![b_1]\!]s = \mathcal{B}[\![b_2]\!]s$$

Show that for each b' of **Bexp′** there exists a boolean expression b of **Bexp** such that b' and b are equivalent. □

1.4 Properties of the semantics

Later in the book we shall be interested in two kinds of properties for expressions. One is that their values do not depend on values of variables that do not occur in them. The other is that if we replace a variable with an expression then we could as well have made a similar change in the state. We shall formalize these properties below and prove that they do hold.

Free variables

The *free variables* of an arithmetic expression a is defined to be the set of variables occurring in it. Formally, we may give a compositional definition of the subset $\text{FV}(a)$ of **Var**:

$$\begin{array}{lcl} \text{FV}(n) & = & \emptyset \\ \text{FV}(x) & = & \{\ x\ \} \\ \text{FV}(a_1 + a_2) & = & \text{FV}(a_1) \cup \text{FV}(a_2) \\ \text{FV}(a_1 \star a_2) & = & \text{FV}(a_1) \cup \text{FV}(a_2) \\ \text{FV}(a_1 - a_2) & = & \text{FV}(a_1) \cup \text{FV}(a_2) \end{array}$$

As an example FV(x+1) = { x } and FV(x+y⋆x) = { x, y }. It should be obvious that only the variables in $\text{FV}(a)$ may influence the value of a. This is formally expressed by:

Lemma 1.11 Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\text{FV}(a)$. Then $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.

Proof: We shall give a fairly detailed proof of the lemma using structural induction on the arithmetic expressions. We shall first consider the basis elements of **Aexp**:

The case n: From Table 1.1 we have $\mathcal{A}[\![n]\!]s = \mathcal{N}[\![n]\!]$ as well as $\mathcal{A}[\![n]\!]s' = \mathcal{N}[\![n]\!]$. So $\mathcal{A}[\![n]\!]s = \mathcal{A}[\![n]\!]s'$ and clearly the lemma holds in this case.

The case x: From Table 1.1 we have $\mathcal{A}[\![x]\!]s = s\ x$ as well as $\mathcal{A}[\![x]\!]s' = s'\ x$. From the assumptions of the lemma we get $s\ x = s'\ x$ because $x \in \text{FV}(x)$ so clearly the lemma holds in this case.

Next we turn to the composite elements of **Aexp**:

The case $a_1 + a_2$: From Table 1.1 we have $\mathcal{A}[\![a_1 + a_2]\!]s = \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![s_2]\!]s$ and similarly $\mathcal{A}[\![a_1 + a_2]\!]s' = \mathcal{A}[\![a_1]\!]s' + \mathcal{A}[\![s_2]\!]s'$. Since a_i (for i = 1,2) is an immediate subexpression of $a_1 + a_2$ and $\text{FV}(a_i) \subseteq \text{FV}(a_1 + a_2)$ we can apply the induction hypothesis (that is the lemma) to a_i and get $\mathcal{A}[\![a_i]\!]s = \mathcal{A}[\![a_i]\!]s'$. It is now easy to see that the lemma holds for $a_1 + a_2$ as well.

The cases $a_1 - a_2$ and $a_1 \star a_2$ follow the same pattern and are omitted. This completes the proof. $\square$

In a similar way we may define the set $\mathrm{FV}(b)$ of free variables in a boolean expression b by

$$\begin{aligned}
\mathrm{FV}(\texttt{true}) &= \emptyset \\
\mathrm{FV}(\texttt{false}) &= \emptyset \\
\mathrm{FV}(a_1 = a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 \leq a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(\neg b) &= \mathrm{FV}(b) \\
\mathrm{FV}(b_1 \wedge b_2) &= \mathrm{FV}(b_1) \cup \mathrm{FV}(b_2)
\end{aligned}$$

Exercise 1.12 (Essential) Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\mathrm{FV}(b)$. Prove that $\mathcal{B}[\![b]\!]s = \mathcal{B}[\![b]\!]s'$. $\square$

Substitutions

We shall later be interested in replacing each occurrence of a variable y in an arithmetic expression a with another arithmetic expression a_0. This is called *substitution* and we write $a[y{\mapsto}a_0]$ for the arithmetic expression so obtained. The formal definition is as follows:

$$\begin{aligned}
n[y{\mapsto}a_0] &= n \\
x[y{\mapsto}a_0] &= \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
(a_1 + a_2)[y{\mapsto}a_0] &= (a_1[y{\mapsto}a_0]) + (a_2[y{\mapsto}a_0]) \\
(a_1 \star a_2)[y{\mapsto}a_0] &= (a_1[y{\mapsto}a_0]) \star (a_2[y{\mapsto}a_0]) \\
(a_1 - a_2)[y{\mapsto}a_0] &= (a_1[y{\mapsto}a_0]) - (a_2[y{\mapsto}a_0])
\end{aligned}$$

As an example $(\mathsf{x}+1)[\mathsf{x}{\mapsto}3] = 3+1$ and $(\mathsf{x}+\mathsf{y}\star\mathsf{x})[\mathsf{x}{\mapsto}\mathsf{y}-5] = (\mathsf{y}-5)+\mathsf{y}\star(\mathsf{y}-5)$.

We also have a notion of substitution (or updating) for states. We define $s[y{\mapsto}v]$ to be the state that is as s except that the value bound to y is v, that is

$$(s[y{\mapsto}v])\ x = \begin{cases} v & \text{if } x = y \\ s\ x & \text{if } x \neq y \end{cases}$$

The relationship between the two concepts is shown in the following exercise:

Exercise 1.13 (Essential) Prove that $\mathcal{A}[\![a[y\mapsto a_0]]\!]s = \mathcal{A}[\![a]\!](s[y\mapsto\mathcal{A}[\![a_0]\!]s])$ for all states s. □

Exercise 1.14 (Essential) Define substitution for boolean expressions: $b[y\mapsto a_0]$ is to be the boolean expression that is as b except that all occurrences of the variable y are replaced by the arithmetic expression a_0. Prove that your definition satisfies

$$\mathcal{B}[\![b[y\mapsto a_0]]\!]s = \mathcal{B}[\![b]\!](s[y\mapsto\mathcal{A}[\![a_0]\!]s])$$

for all states s. □

Chapter 2

Operational Semantics

The role of a statement in **While** is to change the state. For example, if x is bound to **3** in s and we execute the statement x := x + 1 then we get a new state where x is bound to **4**. So while the semantics of arithmetic and boolean expressions only *inspect* the state in order to determine the value of the expression, the semantics of statements will *modify* the state as well.

In an operational semantics we are concerned with *how* to execute programs and not merely what the results of execution are. More precisely, we are interested in how the states are modified during the execution of the statement. We shall consider two different approaches to operational semantics:

- *Natural semantics*: its purpose is to describe how the *overall* results of executions are obtained.
- *Structural operational semantics*: its purpose is to describe how the *individual steps* of the computations take place.

We shall see that for the language **While** we can easily specify both kinds of semantics and that they will be "equivalent" in a sense to be made clear later. However, we shall also give examples of programming constructs where one of the approaches is superior to the other.

For both kinds of operational semantics, the meaning of statements will be specified by a *transition system*. It will have two types of configurations:

$\langle S, s\rangle$	representing that the statement S is to be executed from the state s, and
s	representing a terminal (that is final) state.

The *terminal configurations* will be those of the latter form. The *transition relation* will then describe how the execution takes place. The difference between the two approaches to operational semantics amounts to different ways of specifying the transition relation.

$[\mathrm{ass}_{\mathrm{ns}}]$	$\langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\mathrm{skip}_{\mathrm{ns}}]$	$\langle \texttt{skip},\ s\rangle \to s$
$[\mathrm{comp}_{\mathrm{ns}}]$	$\dfrac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$
$[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}]$	$\dfrac{\langle S_1,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}]$	$\dfrac{\langle S_2,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}]$	$\dfrac{\langle S,\ s\rangle \to s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

Table 2.1: Natural semantics for **While**

2.1 Natural semantics

In a natural semantics we are concerned with the relationship between the *initial* and the *final* state of an execution. Therefore the transition relation will specify the relationship between the initial state and the final state for each statement. We shall write a transition as

$$\langle S,\ s\rangle \to s'$$

Intuitively this means that the execution of S from s will terminate and the resulting state will be s'.

The definition of $\to$ is given by the rules of Table 2.1. A *rule* has the general form

$$\frac{\langle S_1,\ s_1\rangle \to s'_1,\ \cdots,\ \langle S_n,\ s_n\rangle \to s'_n}{\langle S,\ s\rangle \to s'} \ \text{if} \cdots$$

where $S_1, \cdots, S_n$ are *immediate constituents* of S or are statements *constructed from* the immediate constituents of S. A rule has a number of *premises* (written above the solid line) and one *conclusion* (written below the solid line). A rule may also have a number of *conditions* (written to the right of the solid line) that have to be fulfilled whenever the rule is applied. Rules with an empty set of premises are called *axioms* and the solid line is then omitted.

Intuitively, the axiom $[\mathrm{ass}_{\mathrm{ns}}]$ says that in a state s, $x := a$ is executed to yield a final state $s[x \mapsto \mathcal{A}[\![a]\!]s]$ which is as s except that x has the value $\mathcal{A}[\![a]\!]s$. This is really an *axiom schema* because x, a and s are meta-variables standing for arbitrary variables, arithmetic expressions and states but we shall simply use the term axiom for this. We obtain an *instance* of the axiom by selecting particular variables, arithmetic expressions and states. As an example, if s_0 is the state that assigns the value $\mathbf{0}$ to all variables then

$$\langle \texttt{x := x+1},\ s_0\rangle \to s_0[\texttt{x}\mapsto\mathbf{1}]$$

is an instance of [ass$_{\text{ns}}$] because x is instantiated to `x`, a to `x+1`, s to s_0, and the value $\mathcal{A}[\![\texttt{x+1}]\!]s_0$ is determined to be $\mathbf{1}$.

Similarly [skip$_{\text{ns}}$] is an axiom and, intuitively, it says that `skip` does not change the state. Letting s_0 be as above we obtain

$$\langle \texttt{skip},\ s_0\rangle \to s_0$$

as an instance of the axiom [skip$_{\text{ns}}$].

Intuitively, the rule [comp$_{\text{ns}}$] says that to execute S_1;S_2 from state s we must first execute S_1 from s. Assuming that this yields a final state s' we shall then execute S_2 from s'. The premises of the rule are concerned with the two statements S_1 and S_2 whereas the conclusion expresses a property of the composite statement itself. The following is an *instance* of the rule:

$$\frac{\langle \texttt{skip},\ s_0\rangle \to s_0,\ \langle \texttt{x := x+1},\ s_0\rangle \to s_0[\texttt{x}\mapsto\mathbf{1}]}{\langle \texttt{skip; x := x+1},\ s_0\rangle \to s_0[\texttt{x}\mapsto\mathbf{1}]}$$

Here S_1 is instantiated to `skip`, S_2 to `x := x + 1`, s and s' are both instantiated to s_0 and s'' is instantiated to $s_0[\texttt{x}\mapsto\mathbf{1}]$. Similarly

$$\frac{\langle \texttt{skip},\ s_0\rangle \to s_0[\texttt{x}\mapsto\mathbf{5}],\ \langle \texttt{x := x+1},\ s_0[\texttt{x}\mapsto\mathbf{5}]\rangle \to s_0}{\langle \texttt{skip; x := x+1},\ s_0\rangle \to s_0}$$

is an instance of [comp$_{\text{ns}}$] although it is less interesting because its premises can never be derived from the axioms and rules of Table 2.1.

For the `if`-construct we have two rules. The first one, [if$^{\text{tt}}_{\text{ns}}$], says that to execute `if` b `then` S_1 `else` S_2 we simply execute S_1 provided that b evaluates to **tt** in the state. The other rule, [if$^{\text{ff}}_{\text{ns}}$], says that if b evaluates to **ff** then to execute `if` b `then` S_1 `else` S_2 we just execute S_2. Taking s_0 `x` $= \mathbf{0}$ the following is an instance of the rule [if$^{\text{tt}}_{\text{ns}}$]:

$$\frac{\langle \texttt{skip},\ s_0\rangle \to s_0}{\langle \texttt{if x = 0 then skip else x := x+1},\ s_0\rangle \to s_0}$$

because $\mathcal{B}[\![\texttt{x = 0}]\!]s_0 = \textbf{tt}$. However, had it been the case that s_0 `x` $\neq \mathbf{0}$ then it would not be an instance of the rule [if$^{\text{tt}}_{\text{ns}}$] because then $\mathcal{B}[\![\texttt{x = 0}]\!]s_0$ would amount to **ff**. Furthermore it would not be an instance of the rule [if$^{\text{ff}}_{\text{ns}}$] because the premise has the wrong form.

Finally, we have one rule and one axiom expressing how to execute the `while`-construct. Intuitively, the meaning of the construct `while` b `do` S in the state s can be explained as follows:

- If the test b evaluates to true in the state s then we first execute the body of the loop and then continue with the loop itself from the state so obtained.
- If the test b evaluates to false in the state s then the execution of the loop terminates.

The rule [while$^{\text{tt}}_{\text{ns}}$] formalizes the first case where b evaluates to **tt** and it says that then we have to execute S followed by `while` b `do` S again. The axiom [while$^{\text{ff}}_{\text{ns}}$] formalizes the second possibility and states that if b evaluates to **ff** then we terminate the execution of the `while`-construct leaving the state unchanged. Note that the rule [while$^{\text{tt}}_{\text{ns}}$] specifies the meaning of the `while`-construct in terms of the meaning of the very same construct so that we do *not* have a compositional definition of the semantics of statements.

When we use the axioms and rules to derive a transition $\langle S,\ s\rangle \to s'$ we obtain a *derivation tree*. The *root* of the derivation tree is $\langle S,\ s\rangle \to s'$ and the *leaves* are instances of axioms. The *internal nodes* are conclusions of instantiated rules and they have the corresponding premises as their immediate sons. We request that all the instantiated conditions of axioms and rules must be satisfied. When displaying a derivation tree it is common to have the root at the bottom rather than at the top; hence the son is *above* its father. A derivation tree is called *simple* if it is an instance of an axiom, otherwise it is called *composite*.

Example 2.1 Let us first consider the statement of Chapter 1:

(z:=x; x:=y); y:=z

Let s_0 be the state that maps all variables except x and y to **0** and has s_0 x = **5** and s_0 y = **7**. Then the following is an example of a derivation tree:

$$\dfrac{\dfrac{\langle \text{z:=x},\ s_0\rangle \to s_1 \qquad \langle \text{x:=y},\ s_1\rangle \to s_2}{\langle \text{z:=x; x:=y},\ s_0\rangle \to s_2} \qquad \langle \text{y:=z},\ s_2\rangle \to s_3}{\langle \text{(z:=x; x:=y); y:=z},\ s_0\rangle \to s_3}$$

where we have used the abbreviations:

$$\begin{aligned} s_1 &= s_0[\text{z}\mapsto\mathbf{5}] \\ s_2 &= s_1[\text{x}\mapsto\mathbf{7}] \\ s_3 &= s_2[\text{y}\mapsto\mathbf{5}] \end{aligned}$$

The derivation tree has three leaves denoted $\langle$z:=x, $s_0\rangle \to s_1$, $\langle$x:=y, $s_1\rangle \to s_2$, and $\langle$y:=z, $s_2\rangle \to s_3$, corresponding to three applications of the axiom [ass$_{\text{ns}}$]. The rule [comp$_{\text{ns}}$] has been applied twice. One instance is

$$\frac{\langle \text{z:=x}, s_0\rangle \to s_1,\ \langle \text{x:=y}, s_1\rangle \to s_2}{\langle \text{z:=x; x:=y}, s_0\rangle \to s_2}$$

which has been used to combine the leaves $\langle$z:=x, $s_0\rangle \to s_1$ and $\langle$x:=y, $s_1\rangle \to s_2$ with the internal node labelled $\langle$z:=x; x:=y, $s_0\rangle \to s_2$. The other instance is

$$\frac{\langle \text{z:=x; x:=y}, s_0\rangle \to s_2,\ \langle \text{y:=z}, s_2\rangle \to s_3}{\langle \text{(z:=x; x:=y); y:=z}, s_0\rangle \to s_3}$$

which has been used to combine the internal node $\langle$z:=x; x:=y, $s_0\rangle \to s_2$ and the leaf $\langle$y:=z, $s_2\rangle \to s_3$ with the root $\langle$(z:=x; x:=y); y:=z, $s_0\rangle \to s_3$. □

Consider now the problem of constructing a derivation tree for a given statement S and state s. The best way to approach this is to try to construct the tree from the root upwards. So we will start by finding an axiom or rule with a conclusion where the left-hand side matches the configuration $\langle S, s\rangle$. There are two cases:

- If it is an *axiom* and if the conditions of the axiom are satisfied then we can determine the final state and the construction of the derivation tree is completed.
- If it is a *rule* then the next step is to try to construct derivation trees for the premises of the rule. When this has been done, it must be checked that the conditions of the rule are fulfilled, and only then can we determine the final state corresponding to $\langle S, s\rangle$.

Often there will be more than one axiom or rule that matches a given configuration and then the various possibilities have to be inspected in order to find a derivation tree. We shall see later that for **While** there will be at most one derivation tree for each transition $\langle S, s\rangle \to s'$ but that this need not hold in extensions of **While**.

Example 2.2 Consider the factorial statement:

y:=1; `while` ¬(x=1) `do` (y:=y ⋆ x; x:=x−1)

and let s be a state with s x = **3**. In this example we shall show that

$$\langle \text{y:=1; while } \neg(\text{x=1}) \text{ do (y:=y} \star \text{x; x:=x}-1), s\rangle \to s[\text{y}\mapsto \mathbf{6}][\text{x}\mapsto \mathbf{1}] \qquad (*)$$

To do so we shall show that (*) can be obtained from the transition system of Table 2.1. This is done by constructing a derivation tree with the transition (*) as its root.

Rather than presenting the complete derivation tree T in one go, we shall build it in an upwards manner. Initially, we only know that the root of T is of the form:

$$\langle \texttt{y:=1; while } \neg(\texttt{x=1}) \texttt{ do (y:=y} \star \texttt{x; x:=x}-1\texttt{)},\ s\rangle \to s_{61}$$

However, the statement

$$\texttt{y:=1; while } \neg(\texttt{x=1}) \texttt{ do (y:=y} \star \texttt{x; x:=x}-1\texttt{)}$$

is of the form S_1; S_2 so the only rule that could have been used to produce the root of T is [comp$_{\text{ns}}$]. Therefore T must have the form:

$$\frac{\langle \texttt{y:=1},\ s\rangle \to s_{13} \qquad\qquad T_1}{\langle \texttt{y:=1; while } \neg(\texttt{x=1}) \texttt{ do (y:=y}\star\texttt{x; x:=x}-1\texttt{)},\ s\rangle \to s_{61}}$$

for some state s_{13} and some derivation tree T_1 which has root

$$\langle \texttt{while } \neg(\texttt{x=1}) \texttt{ do (y:=y}\star\texttt{x; x:=x}-1\texttt{)},\ s_{13}\rangle \to s_{61} \qquad (**)$$

Since $\langle \texttt{y:=1},\ s\rangle \to s_{13}$ has to be an instance of the axiom [ass$_{\text{ns}}$] we get that $s_{13} = s[\texttt{y}\mapsto\mathbf{1}]$.

The missing part T_1 of T is a derivation tree with root (**). Since the statement of (**) has the form `while` b `do` S the derivation tree T_1 must have been constructed by applying either the rule [while$^{\text{tt}}_{\text{ns}}$] or the axiom [while$^{\text{ff}}_{\text{ns}}$]. Since $\mathcal{B}[\![\neg(\texttt{x=1})]\!]s_{13} = \mathbf{tt}$ we see that only the rule [while$^{\text{tt}}_{\text{ns}}$] could have been applied so T_1 will have the form:

$$\frac{T_2 \qquad\qquad T_3}{\langle \texttt{while } \neg(\texttt{x=1}) \texttt{ do (y:=y}\star\texttt{x; x:=x}-1\texttt{)},\ s_{13}\rangle \to s_{61}}$$

where T_2 is a derivation tree with root

$$\langle \texttt{y:=y}\star\texttt{x; x:=x}-1,\ s_{13}\rangle \to s_{32}$$

and T_3 is a derivation tree with root

$$\langle \texttt{while } \neg(\texttt{x=1}) \texttt{ do (y:=y}\star\texttt{x; x:=x}-1\texttt{)},\ s_{32}\rangle \to s_{61} \qquad (***)$$

for some state s_{32}.

Using that the form of the statement y:=y⋆x; x:=x−1 is S_1;S_2 it is now easy to see that the derivation tree T_2 is

$$\frac{\langle \texttt{y:=y}\star\texttt{x},\ s_{13}\rangle \to s_{33} \qquad \langle \texttt{x:=x}-1,\ s_{33}\rangle \to s_{32}}{\langle \texttt{y:=y}\star\texttt{x; x:=x}-1,\ s_{13}\rangle \to s_{32}}$$

where $s_{33} = s[\texttt{y}\mapsto\mathbf{3}]$ and $s_{32} = s[\texttt{y}\mapsto\mathbf{3}][\texttt{x}\mapsto\mathbf{2}]$. The leaves of T_2 are instances of [ass$_{\text{ns}}$] and they are combined using [comp$_{\text{ns}}$]. So now T_2 is fully constructed.

In a similar way we can construct the derivation tree T_3 with root (***) and we get:

$$\dfrac{\dfrac{\langle \texttt{y:=y}\star\texttt{x},\ s_{32}\rangle\to s_{62} \qquad \langle \texttt{x:=x}-1,\ s_{62}\rangle\to s_{61}}{\langle \texttt{y:=y}\star\texttt{x; x:=x}-1,\ s_{32}\rangle\to s_{61}} \qquad T_4}{\langle \texttt{while}\ \neg(\texttt{x}=1)\ \texttt{do (y:=y}\star\texttt{x; x:=x}-1),\ s_{32}\rangle\to s_{61}}$$

where $s_{62} = s[\texttt{y}\mapsto\mathbf{6}][\texttt{x}\mapsto\mathbf{2}]$, $s_{61} = s[\texttt{y}\mapsto\mathbf{6}][\texttt{x}\mapsto\mathbf{1}]$ and T_4 is a derivation tree with root

$$\langle \texttt{while}\ \neg(\texttt{x}=1)\ \texttt{do (y:=y}\star\texttt{x; x:=x}-1),\ s_{61}\rangle\to s_{61}$$

Finally, we see that the derivation tree T_4 is an instance of the axiom $[\text{while}^{\text{ff}}_{\text{ns}}]$ because $\mathcal{B}[\![\neg(\texttt{x}=1)]\!]s_{61} = \mathbf{ff}$. This completes the construction of the derivation tree T for (*). □

Exercise 2.3 Consider the statement

$$\texttt{z:=0; while y}\leq\texttt{x do (z:=z+1; x:=x}-\texttt{y)}$$

Construct a derivation tree for this statement when executed in a state where x has the value **17** and y has the value **5**. □

We shall introduce the following terminology: The execution of a statement S on a state s

- *terminates* if and only if there is a state s' such that $\langle S,\ s\rangle \to s'$, and
- *loops* if and only if there is *no* state s' such that $\langle S,\ s\rangle \to s'$.

We shall say that a statement S *always terminates* if its execution on a state s terminates for all choices of s, and *always loops* if its execution on a state s loops for all choices of s.

Exercise 2.4 Consider the following statements

- $\texttt{while}\ \neg(\texttt{x}=1)\ \texttt{do (y:=y}\star\texttt{x; x:=x}-1)$
- $\texttt{while}\ 1\leq\texttt{x do (y:=y}\star\texttt{x; x:=x}-1)$
- `while true do skip`

For each statement determine whether or not it always terminates and whether or not it always loops. Try to argue for your answers using the axioms and rules of Table 2.1. □

Properties of the semantics

The transition system gives us a way of arguing about statements and their properties. As an example we may be interested in whether two statements S_1 and S_2 are *semantically equivalent*; by this we mean that for all states s and s'

$$\langle S_1, s\rangle \to s' \text{ if and only if } \langle S_2, s\rangle \to s'$$

Lemma 2.5 The statement

$$\texttt{while } b \texttt{ do } S$$

is semantically equivalent to

$$\texttt{if } b \texttt{ then } (S;\ \texttt{while } b \texttt{ do } S) \texttt{ else skip}.$$

Proof: The proof is in two stages. We shall first prove that if

$$\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s'' \qquad (*)$$

then

$$\langle \texttt{if } b \texttt{ then } (S;\ \texttt{while } b \texttt{ do } S) \texttt{ else skip},\ s\rangle \to s'' \qquad (**)$$

Thus, if the execution of the loop terminates then so does its one-level unfolding. Later we shall show that if the unfolded loop terminates then so will the loop itself; the conjunction of these results then prove the lemma.

Because (*) holds we know that we have a derivation tree T for it. It can have one of two forms depending on whether it has been constructed using the rule [while$^{\text{tt}}_{\text{ns}}$] or the axiom [while$^{\text{ff}}_{\text{ns}}$]. In the first case the derivation tree T has the form:

$$\frac{T_1 \qquad T_2}{\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s''}$$

where T_1 is a derivation tree with root $\langle S, s\rangle \to s'$ and T_2 is a derivation tree with root $\langle \texttt{while } b \texttt{ do } S,\ s'\rangle \to s''$. Furthermore, $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Using the derivation trees T_1 and T_2 as the premises for the rules [comp$_{\text{ns}}$] we can construct the derivation tree:

$$\frac{T_1 \qquad T_2}{\langle S;\ \texttt{while } b \texttt{ do } S,\ s\rangle \to s''}$$

Using that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ we can use the rule [if$^{\text{tt}}_{\text{ns}}$] to construct the derivation tree

$$\cfrac{\cfrac{T_1 \qquad\qquad\qquad T_2}{\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}}{\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle \to s''}$$

thereby showing that (**) holds.

Alternatively, the derivation tree T is an instance of [while$^{\text{ff}}_{\text{ns}}$]. Then $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and we must have that s''=s. So T simply is

$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s$

Using the axiom [skip$_{\text{ns}}$] we get a derivation tree

$\langle \texttt{skip},\ s\rangle{\to}s''$

and we can now apply the rule [if$^{\text{ff}}_{\text{ns}}$] to construct a derivation tree for (**):

$$\cfrac{\langle \texttt{skip},\ s\rangle \to s''}{\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle \to s''}$$

This completes the first part of the proof.

For the second stage of the proof we assume that (**) holds and shall prove that (*) holds. So we have a derivation tree T for (**) and must construct one for (*). Only two rules could give rise to the derivation tree T for (**), namely [if$^{\text{tt}}_{\text{ns}}$] or [if$^{\text{ff}}_{\text{ns}}$]. In the first case, $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and we have a derivation tree T_1 with root

$\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle{\to}s''$

The statement has the general form S_1; S_2 and the only rule that could give this is [comp$_{\text{ns}}$]. Therefore there are derivation trees T_2 and T_3 for

$\langle S,\ s\rangle{\to}s'$, and

$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle{\to}s''$

for some state s'. It is now straightforward to use the rule [while$^{\text{tt}}_{\text{ns}}$] to combine T_2 and T_3 to a derivation tree for (*).

In the second case, $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and T is constructed using the rule [if$^{\text{ff}}_{\text{ns}}$]. This means that we have a derivation tree for

$\langle \texttt{skip},\ s\rangle{\to}s''$

and according to axiom [skip$_{\text{ns}}$] it must be the case that s=s''. But then we can use the axiom [while$^{\text{ff}}_{\text{ns}}$] to construct a derivation tree for (*). This completes the proof. □

Exercise 2.6 Prove that the two statements S_1;(S_2;S_3) and (S_1;S_2);S_3 are semantically equivalent. Construct a statement showing that S_1;S_2 is not, in general, semantically equivalent to S_2;S_1. □

Exercise 2.7 Extend the language **While** with the statement

`repeat` S `until` b

and define the relation → for it. (The semantics of the `repeat`-construct is not allowed to rely on the existence of a `while`-construct in the language.) Prove that `repeat` S `until` b and S; `if` b `then skip else (repeat` S `until` b`)` are semantically equivalent. □

Exercise 2.8 Another iterative construct is

`for` x := a_1 `to` a_2 `do` S

Extend the language **While** with this statement and define the relation → for it. Evaluate the statement

`y:=1; for z:=1 to x do (y:=y ⋆ x; x:=x−1)`

from a state where `x` has the value 5. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$, so that there is a numeral for each number that may arise during the computation. (The semantics of the `for`-construct is not allowed to rely on the existence of a `while`-construct in the language.) □

In the above proof we used Table 2.1 to inspect the structure of the derivation tree for a certain transition known to hold. In the proof of the next result we shall combine this with an *induction on the shape of the derivation tree*. The idea can be summarized as follows:

Induction on the Shape of Derivation Trees

1: Prove that the property holds for all the simple derivation trees by showing that it holds for the *axioms* of the transition system.

2: Prove that the property holds for all composite derivation trees: For each *rule* assume that the property holds for its premises (this is called the *induction hypothesis*) and prove that it also holds for the conclusion of the rule provided that the conditions of the rule are satisfied.

To formulate the theorem we shall say that the semantics of Table 2.1 is *deterministic* if for all choices of S, s, s' and s'' we have that

$\langle S, s\rangle \rightarrow s'$ and $\langle S, s\rangle \rightarrow s''$ imply $s' = s''$

This means that for every statement S and initial state s we can uniquely determine a final state s' if (and only if) the execution of S terminates.

Theorem 2.9 The natural semantics of Table 2.1 is deterministic.

Proof: We assume that $\langle S,\ s\rangle \rightarrow s'$ and shall prove that

$$\text{if } \langle S,\ s\rangle \rightarrow s'' \text{ then } s' = s''.$$

We shall proceed by induction on the shape of the derivation tree for $\langle S,\ s\rangle \rightarrow s'$.

The case [ass$_{\text{ns}}$]: Then S is $x{:=}a$ and s' is $s[x \mapsto \mathcal{A}[\![a]\!]s]$. The only axiom or rule that could be used to give $\langle x{:=}a,\ s\rangle \rightarrow s''$ is [ass$_{\text{ns}}$] so it follows that s'' must be $s[x \mapsto \mathcal{A}[\![a]\!]s]$ and thereby $s' = s''$.

The case [skip$_{\text{ns}}$]: Analogous.

The case [comp$_{\text{ns}}$]: Assume that

$$\langle S_1{;}S_2,\ s\rangle \rightarrow s'$$

holds because

$$\langle S_1,\ s\rangle \rightarrow s_0 \text{ and } \langle S_2,\ s_0\rangle \rightarrow s'$$

for some s_0. The only rule that could be applied to give $\langle S_1{;}S_2,\ s\rangle \rightarrow s''$ is [comp$_{\text{ns}}$] so there is a state s_1 such that

$$\langle S_1,\ s\rangle \rightarrow s_1 \text{ and } \langle S_2,\ s_1\rangle \rightarrow s''$$

The induction hypothesis can be applied to the premise $\langle S_1,\ s\rangle \rightarrow s_0$ and from $\langle S_1,\ s\rangle \rightarrow s_1$ we get $s_0 = s_1$. Similarly, the induction hypothesis can be applied to the premise $\langle S_2,\ s_0\rangle \rightarrow s'$ and from $\langle S_2,\ s_0\rangle \rightarrow s''$ we get $s' = s''$ as required.

The case [if$^{\text{tt}}_{\text{ns}}$]: Assume that

$$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\ s\rangle \rightarrow s'$$

holds because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \langle S_1,\ s\rangle \rightarrow s'$$

From $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ we get that the only rule that could be applied to give the alternative $\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\ s\rangle \rightarrow s''$ is [if$^{\text{tt}}_{\text{ns}}$]. So it must be the case that

$$\langle S_1,\ s\rangle \rightarrow s''$$

But then the induction hypothesis can be applied to the premise $\langle S_1, s\rangle \to s'$ and from $\langle S_1, s\rangle \to s''$ we get $s' = s''$.

The case $[\text{if}_{\text{ns}}^{\text{ff}}]$: Analogous.

The case $[\text{while}_{\text{ns}}^{\text{tt}}]$: Assume that

$$\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s'$$

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt},\ \langle S,\ s\rangle{\to}s_0 \text{ and } \langle \texttt{while } b \texttt{ do } S,\ s_0\rangle{\to}s'$$

The only rule that could be applied to give $\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s''$ is $[\text{while}_{\text{ns}}^{\text{tt}}]$ because $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and this means that

$$\langle S,\ s\rangle{\to}s_1 \text{ and } \langle \texttt{while } b \texttt{ do } S,\ s_1\rangle \to s''$$

must hold for some s_1. Again the induction hypothesis can be applied to the premise $\langle S,\ s\rangle{\to}s_0$ and from $\langle S,\ s\rangle{\to}s_1$ we get $s_0 = s_1$. Thus we have

$$\langle \texttt{while } b \texttt{ do } S,\ s_0\rangle{\to}s' \text{ and } \langle \texttt{while } b \texttt{ do } S,\ s_0\rangle{\to}s''$$

Since $\langle \texttt{while } b \texttt{ do } S,\ s_0\rangle{\to}s'$ is a premise of (the instance of) $[\text{while}_{\text{ns}}^{\text{tt}}]$ we can apply the induction hypothesis to it. From $\langle \texttt{while } b \texttt{ do } S,\ s_0\rangle{\to}s''$ we therefore get $s' = s''$ as required.

The case $[\text{while}_{\text{ns}}^{\text{ff}}]$: Straightforward. □

Exercise 2.10 * Prove that $\texttt{repeat } S \texttt{ until } b$ (as defined in Exercise 2.7) is semantically equivalent to S; $\texttt{while } \neg b \texttt{ do } S$. Argue that this means that the extended semantics is deterministic. □

It is worth observing that we could not prove Theorem 2.9 using structural induction on the statement S. The reason is that the rule $[\text{while}_{\text{ns}}^{\text{tt}}]$ defines the semantics of $\texttt{while } b \texttt{ do } S$ in terms of itself. Structural induction works fine when the semantics is defined *compositionally* (as e.g. $\mathcal{A}$ and $\mathcal{B}$ in Chapter 1). But the natural semantics of Table 2.1 is *not* defined compositionally because of the rule $[\text{while}_{\text{ns}}^{\text{tt}}]$.

Basically, induction on the shape of derivation trees is a kind of structural induction on the derivation trees: In the *base case* we show that the property holds for the simple derivation trees. In the *induction step* we assume that the property holds for the immediate constituents of a derivation tree and show that it also holds for the composite derivation tree.

The semantic function $\mathcal{S}_{\mathrm{ns}}$

The *meaning* of statements can now be summarized as a (partial) function from **State** to **State**. We define

$$\mathcal{S}_{\mathrm{ns}}\text{: } \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

and this means that for every statement S we have a partial function

$$\mathcal{S}_{\mathrm{ns}}[\![S]\!] \in \mathbf{State} \hookrightarrow \mathbf{State}.$$

It is given by

$$\mathcal{S}_{\mathrm{ns}}[\![S]\!]s = \begin{cases} s' & \text{if } \langle S, s\rangle \to s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

Note that $\mathcal{S}_{\mathrm{ns}}$ is a well-defined partial function because of Theorem 2.9. The need for partiality is demonstrated by the statement `while true do skip` that always loops (see Exercise 2.4); we then have

$$\mathcal{S}_{\mathrm{ns}}[\![\texttt{while true do skip}]\!]\ s = \underline{\text{undef}}$$

for all states s.

Exercise 2.11 The semantics of arithmetic expressions is given by the function $\mathcal{A}$. We can also use an operational approach and define a natural semantics for the arithmetic expressions. It will have two kinds of configurations:

$\langle a, s\rangle$	denoting that a has to be evaluated in state s, and
z	denoting the final value (an element of $\mathbf{Z}$).

The transition relation $\to_{\mathrm{Aexp}}$ has the form

$$\langle a, s\rangle \to_{\mathrm{Aexp}} z$$

where the idea is that a evaluates to z in state s. Some example axioms and rules are

$$\langle n, s\rangle \to_{\mathrm{Aexp}} \mathcal{N}[\![n]\!]$$

$$\langle x, s\rangle \to_{\mathrm{Aexp}} s\ x$$

$$\frac{\langle a_1, s\rangle \to_{\mathrm{Aexp}} z_1,\ \langle a_2, s\rangle \to_{\mathrm{Aexp}} z_2}{\langle a_1 + a_2, s\rangle \to_{\mathrm{Aexp}} z} \quad \text{where } z = z_1 + z_2$$

Complete the specification of the transition system. Use structural induction on **Aexp** to prove that the meaning of a defined by this relation is the same as that defined by $\mathcal{A}$. □

Exercise 2.12 In a similar way we can specify a natural semantics for the boolean expressions. The transitions will have the form

$$\langle b,\, s\rangle \to_{\mathrm{Bexp}} t$$

where $t \in \mathbf{T}$. Specify the transition system and prove that the meaning of b defined in this way is the same as that defined by $\mathcal{B}$. □

Exercise 2.13 Determine whether or not semantic equivalence of S_1 and S_2 amounts to $\mathcal{S}_{\mathrm{ns}}[\![S_1]\!] = \mathcal{S}_{\mathrm{ns}}[\![S_2]\!]$. □

2.2 Structural operational semantics

In structural operational semantics the emphasis is on the *individual steps* of the execution, that is the execution of assignments and tests. The transition relation has the form

$$\langle S,\, s\rangle \Rightarrow \gamma$$

where γ either is of the form $\langle S',\, s'\rangle$ or of the form s'. The transition expresses the *first* step of the execution of S from state s. There are two possible outcomes:

- If γ is of the form $\langle S',\, s'\rangle$ then the execution of S from s is *not* completed and the remaining computation is expressed by the intermediate configuration $\langle S',\, s'\rangle$.
- If γ is of the form s' then the execution of S from s *has* terminated and the final state is s'.

We shall say that $\langle S,\, s\rangle$ is *stuck* if there is no γ such that $\langle S,\, s\rangle \Rightarrow \gamma$.

The definition of $\Rightarrow$ is given by the axioms and rules of Table 2.2 and the general form of these are as in the previous section. Axioms [ass$_{\mathrm{sos}}$] and [skip$_{\mathrm{sos}}$] have not changed at all because the assignment and `skip` statements are fully executed in one step.

The rules [comp$^1_{\mathrm{sos}}$] and [comp$^2_{\mathrm{sos}}$] express that to execute S_1;S_2 in state s we first execute S_1 one step from s. Then there are two possible outcomes:

- If the execution of S_1 has not been completed we have to complete it before embarking on the execution of S_2.
- If the execution of S_1 has been completed we can start on the execution of S_2.

[ass$_{\text{sos}}$]	$\langle x := a,\ s\rangle \Rightarrow s[x\mapsto\mathcal{A}[\![a]\!]s]$
[skip$_{\text{sos}}$]	$\langle \texttt{skip},\ s\rangle \Rightarrow s$
[comp$^1_{\text{sos}}$]	$\dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$
[comp$^2_{\text{sos}}$]	$\dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$
[if$^{\text{tt}}_{\text{sos}}$]	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
[if$^{\text{ff}}_{\text{sos}}$]	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
[while$_{\text{sos}}$]	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow$ $\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle$

Table 2.2: Structural operational semantics for **While**

The first case is captured by the rule [comp$^1_{\text{sos}}$]: If the result of executing the first step of $\langle S,\ s\rangle$ is an intermediate configuration $\langle S_1',\ s'\rangle$ then the next configuration is $\langle S_1';S_2,\ s'\rangle$ showing that we have to complete the execution of S_1 before we can start on S_2. The second case above is captured by the rule [comp$^2_{\text{sos}}$]: If the result of executing S_1 from s is a final state s' then the next configuration is $\langle S_2,\ s'\rangle$, so that we can now start on S_2.

From the axioms [if$^{\text{tt}}_{\text{sos}}$] and [if$^{\text{ff}}_{\text{sos}}$] we see that the first step in executing a conditional is to perform the test and to select the appropriate branch. Finally, the axiom [while$_{\text{sos}}$] shows that the first step in the execution of the `while`-construct is to unfold it one level, that is to rewrite it as a conditional. The test will therefore be performed in the second step of the execution (where one of the axioms for the `if`-construct is applied). We shall see an example of this shortly.

A *derivation sequence* of a statement S starting in state s is either

- a *finite* sequence

  $$\gamma_0,\ \gamma_1,\ \gamma_2,\ \cdots,\ \gamma_{\text{k}}$$

  of configurations satisfying $\gamma_0 = \langle S,\ s\rangle$, $\gamma_{\text{i}} \Rightarrow \gamma_{\text{i+1}}$ for $0\leq\text{i}<\text{k}$, $\text{k}\geq 0$, and where γ_{k} is either a terminal configuration or a stuck configuration, or it is

- an *infinite* sequence

  $$\gamma_0,\ \gamma_1,\ \gamma_2,\ \cdots$$

of configurations satisfying $\gamma_0 = \langle S, s\rangle$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for $0 \leq i$

We shall write $\gamma_0 \Rightarrow^i \gamma_i$ to indicate that there are i steps in the execution from γ_0 to γ_i and we write $\gamma_0 \Rightarrow^* \gamma_i$ to indicate that there is a finite number of steps. Note that $\gamma_0 \Rightarrow^i \gamma_i$ and $\gamma_0 \Rightarrow^* \gamma_i$ need *not* be derivation sequences: they will be so if and only if γ_i is either a terminal configuration or a stuck configuration.

Example 2.14 Consider the statement

$$(\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z}$$

of Chapter 1 and let s_0 be the state that maps all variables except x and y to $\mathbf{0}$ and that has $s_0\ \mathtt{x} = \mathbf{5}$ and $s_0\ \mathtt{y} = \mathbf{7}$. We then have the derivation sequence:

$$\begin{aligned}
&\langle(\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0\rangle \\
&\qquad \Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle \\
&\qquad \Rightarrow \langle \mathtt{y} := \mathtt{z},\ (s_0[\mathtt{z}\mapsto\mathbf{5}])[\mathtt{x}\mapsto\mathbf{7}]\rangle \\
&\qquad \Rightarrow ((s_0[\mathtt{z}\mapsto\mathbf{5}])[\mathtt{x}\mapsto\mathbf{7}])[\mathtt{y}\mapsto\mathbf{5}]
\end{aligned}$$

Corresponding to *each* of these steps we have *derivation trees* explaining why they take place. For the first step

$$\langle(\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0\rangle \Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle$$

the derivation tree is

$$\dfrac{\dfrac{\langle \mathtt{z} := \mathtt{x},\ s_0\rangle \Rightarrow s_0[\mathtt{z}\mapsto\mathbf{5}]}{\langle \mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y},\ s_0\rangle \Rightarrow \langle \mathtt{x} := \mathtt{y},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle}}{\langle(\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0\rangle \Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle}$$

and it has been constructed from the axiom $[\text{ass}_{\text{sos}}]$ and the rules $[\text{comp}^1_{\text{sos}}]$ and $[\text{comp}^2_{\text{sos}}]$. The derivation tree for the second step is constructed in a similar way using only $[\text{ass}_{\text{sos}}]$ and $[\text{comp}^2_{\text{sos}}]$ and for the third step it simply is an instance of $[\text{ass}_{\text{sos}}]$. □

Example 2.15 Assume that $s\ \mathtt{x} = \mathbf{3}$. The first step of execution from the configuration

$$\langle \mathtt{y}{:=}1;\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y}{:=}\mathtt{y} \star \mathtt{x};\ \mathtt{x}{:=}\mathtt{x}{-}1),\ s\rangle$$

will give the configuration

$$\langle \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y}{:=}\mathtt{y} \star \mathtt{x};\ \mathtt{x}{:=}\mathtt{x}{-}1),\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle$$

This is achieved using the axiom [ass$_{\rm sos}$] and the rule [comp$^2_{\rm sos}$] as shown by the derivation tree:

$$\frac{\langle \texttt{y:=1},\ s\rangle \Rightarrow s[\texttt{y}\mapsto\mathbf{1}]}{\begin{array}{l}\langle \texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s\rangle \Rightarrow\\ \quad\langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\end{array}}$$

The next step of the execution will rewrite the loop as a conditional using the axiom [while$_{\rm sos}$] so we get the configuration

$$\begin{array}{l}\langle \texttt{if}\ \neg(\texttt{x=1})\ \texttt{then ((y:=y}\star\texttt{x; x:=x}-\texttt{1);}\\ \qquad\qquad\qquad\qquad \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1))}\\ \qquad\qquad\quad \texttt{else skip},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\end{array}$$

The following step will perform the test and yields (according to [if$^{\rm tt}_{\rm sos}$]) the configuration

$$\langle \texttt{(y:=y}\star\texttt{x; x:=x}-\texttt{1); while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle$$

We can then use [ass$_{\rm sos}$], [comp$^2_{\rm sos}$] and [comp$^1_{\rm sos}$] to obtain the configuration

$$\langle \texttt{x:=x}-\texttt{1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{3}]\rangle$$

as is verified by the derivation tree:

$$\frac{\dfrac{\langle \texttt{y:=y}\star\texttt{x},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\Rightarrow s[\texttt{y}\mapsto\mathbf{3}]}{\langle \texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\Rightarrow\langle \texttt{x:=x}-\texttt{1},\ s[\texttt{y}\mapsto\mathbf{3}]\rangle}}{\begin{array}{l}\langle \texttt{(y:=y}\star\texttt{x; x:=x}-\texttt{1); while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle \Rightarrow\\ \qquad\langle \texttt{x:=x}-\texttt{1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{3}]\rangle\end{array}}$$

Using [ass$_{\rm sos}$] and [comp$^2_{\rm sos}$] the next configuration will then be

$$\langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{3}][\texttt{x}\mapsto\mathbf{2}]\rangle$$

Continuing in this way we eventually reach the final state $s[\texttt{y}\mapsto\mathbf{6}][\texttt{x}\mapsto\mathbf{1}]$. □

Exercise 2.16 Construct a derivation sequence for the statement

$$\texttt{z:=0; while y}\leq\texttt{x do (z:=z+1; x:=x}-\texttt{y)}$$

when executed in a state where x has the value **17** and y has the value **5**. Determine a state s such that the derivation sequence obtained for the above statement and s is infinite. □

Given a statement S in the language **While** and a state s it is always possible to find *at least one* derivation sequence that starts in the configuration $\langle S, s\rangle$: simply apply axioms and rules forever or until a terminal or stuck configuration is reached. Inspection of Table 2.2 shows that there are no stuck configurations in **While** and Exercise 2.22 below will show that there is in fact only one derivation sequence that starts with $\langle S, s\rangle$. However, some of the constructs considered in Section 2.4 that extend **While** will have configurations that are stuck or more than one derivation sequence that starts in a given configuration.

In analogy with the terminology of the previous section we shall say that the execution of a statement S on a state s

- *terminates* if and only if there is a finite derivation sequence starting with $\langle S, s\rangle$, and
- *loops* if and only if there is an infinite derivation sequence starting with $\langle S, s\rangle$.

We shall say that the execution of S on s *terminates successfully* if $\langle S, s\rangle \Rightarrow^* s'$ for some state s'; in **While** an execution terminates successfully if and only if it terminates because there are no stuck configurations. Finally, we shall say that a statement S *always terminates* if it terminates on all states, and *always loops* if it loops on all states.

Exercise 2.17 Extend **While** with the construct `repeat` S `until` b and specify the structural operational semantics for it. (The semantics for the `repeat`-construct is not allowed to rely on the existence of a `while`-construct.) □

Exercise 2.18 Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify the structural operational semantics for it. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$, so that there is a numeral for each number that may arise during the computation. (The semantics for the `for`-construct is not allowed to rely on the existence of a `while`-construct.) □

Properties of the semantics

For structural operational semantics it is often useful to conduct proofs by induction on the *length* of the derivation sequences. The proof technique may be summarized as follows:

Induction on the Length of Derivation Sequences	
1:	Prove that the property holds for all derivation sequences of length 0.
2:	Prove that the property holds for all other derivation sequences: Assume that the property holds for all derivation sequences of length at most k (this is called the *induction hypothesis*) and show that it holds for derivation sequences of length k+1.

The induction step of a proof following this principle will often be done by inspecting either

- the structure of the syntactic element, or
- the derivation tree validating the first transition of the derivation sequence.

Note that the proof technique is a simple application of mathematical induction.

To illustrate the use of the proof technique we shall prove the following lemma (to be used in the next section). Intuitively, the lemma expresses that the execution of a composite construct $S_1;S_2$ can be split into two parts, one corresponding to S_1 and the other corresponding to S_2.

Lemma 2.19 If $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}} s''$ then there exists a state s' and natural numbers k_1 and k_2 such that $\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1} s'$ and $\langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_2} s''$ where $\mathrm{k} = \mathrm{k}_1+\mathrm{k}_2$.

Proof: The proof is by induction on the number k, that is by induction on the length of the derivation sequence $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}} s''$.

If $\mathrm{k} = 0$ then the result holds vacuously.

For the induction step we assume that the lemma holds for $\mathrm{k} \leq \mathrm{k}_0$ and we shall prove it for k_0+1. So assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}_0+1} s''$$

This means that the derivation sequence can be written as

$$\langle S_1;S_2,\ s\rangle \Rightarrow \gamma \Rightarrow^{\mathrm{k}_0} s''$$

for some configuration γ. Now one of two cases applies depending on which of the two rules $[\mathrm{comp}^1_{\mathrm{sos}}]$ and $[\mathrm{comp}^2_{\mathrm{sos}}]$ was used to obtain $\langle S_1;S_2,\ s\rangle \Rightarrow \gamma$.

In the first case where $[\mathrm{comp}^1_{\mathrm{sos}}]$ is used we have

$$\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle$$

because

$$\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle$$

We therefore have

$$\langle S_1';S_2, s'\rangle \Rightarrow^{\mathrm{k}_0} s''$$

and the induction hypothesis can be applied to this derivation sequence because it is shorter than the one we started with. This means that there is a state s_0 and natural numbers k_1 and k_2 such that

$$\langle S_1', s'\rangle \Rightarrow^{\mathrm{k}_1} s_0 \text{ and } \langle S_2, s_0\rangle \Rightarrow^{\mathrm{k}_2} s''$$

where k_1+k_2=k_0. Using that $\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle$ and $\langle S_1', s'\rangle \Rightarrow^{\mathrm{k}_1} s_0$ we get

$$\langle S_1, s\rangle \Rightarrow^{\mathrm{k}_1+1} s_0$$

We have already seen that $\langle S_2, s_0\rangle \Rightarrow^{\mathrm{k}_2} s''$ and since (k_1+1)+k_2 = k_0+1 we have proved the required result.

The second possibility is that [$\mathrm{comp}^2_{\mathrm{sos}}$] has been used to obtain the derivation $\langle S_1;S_2, s\rangle \Rightarrow \gamma$. Then we have

$$\langle S_1, s\rangle \Rightarrow s'$$

and γ is $\langle S_2, s'\rangle$ so that

$$\langle S_2, s'\rangle \Rightarrow^{\mathrm{k}_0} s''$$

The result now follows by choosing k_1=1 and k_2=k_0. □

Exercise 2.20 Suppose that $\langle S_1;S_2, s\rangle \Rightarrow^* \langle S_2, s'\rangle$. Show that it is *not* necessarily the case that $\langle S_1, s\rangle \Rightarrow^* s'$. □

Exercise 2.21 (Essential) Prove that

$$\text{if } \langle S_1, s\rangle \Rightarrow^{\mathrm{k}} s' \text{ then } \langle S_1;S_2, s\rangle \Rightarrow^{\mathrm{k}} \langle S_2, s'\rangle$$

that is the execution of S_1 is not influenced by the statement following it. □

In the previous section we defined a notion of determinism based on the natural semantics. For the structural operational semantics we define the similar notion as follows. The semantics of Table 2.2 is *deterministic* if for all choices of S, s, γ and γ' we have that

$$\langle S, s\rangle \Rightarrow \gamma \text{ and } \langle S, s\rangle \Rightarrow \gamma' \text{ imply } \gamma = \gamma'$$

Exercise 2.22 (Essential) Show that the structural operational semantics of Table 2.2 is deterministic. Deduce that there is exactly one derivation sequence starting in a configuration $\langle S, s\rangle$. Argue that a statement S of **While** cannot both terminate and loop on a state s and hence it cannot both be always terminating and always looping. □

In the previous section we defined a notion of two statements S_1 and S_2 being semantically equivalent. The similar notion can be defined based on the structural operational semantics: S_1 and S_2 are *semantically equivalent* if for all states s

- $\langle S_1, s\rangle \Rightarrow^* \gamma$ if and only if $\langle S_2, s\rangle \Rightarrow^* \gamma$, whenever γ is a configuration that is either stuck or terminal, and
- there is an infinite derivation sequence starting in $\langle S_1, s\rangle$ if and only if there is one starting in $\langle S_2, s\rangle$.

Note that in the first case the length of the two derivation sequences may be different.

Exercise 2.23 Show that the following statements of **While** are semantically equivalent in the above sense:

- S;`skip` and S
- `while` b `do` S and `if` b `then` (S; `while` b `do` S) `else skip`
- S_1;(S_2;S_3) and (S_1;S_2);S_3

You may use the result of Exercise 2.22. Discuss to what extent the notion of semantic equivalence introduced above is the same as that defined from the natural semantics. □

Exercise 2.24 Prove that `repeat` S `until` b (as defined in Exercise 2.17) is semantically equivalent to S; `while` $\neg$ b `do` S. □

The semantic function $\mathcal{S}_{\text{sos}}$

As in the previous section the *meaning* of statements can be summarized by a (partial) function from **State** to **State**:

$$\mathcal{S}_{\text{sos}}\colon \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

It is given by

$$\mathcal{S}_{\text{sos}}[\![S]\!]s = \begin{cases} s' & \text{if } \langle S, s\rangle \Rightarrow^* s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

The well-definedness of the definition follows from Exercise 2.22.

Exercise 2.25 Determine whether or not semantic equivalence of S_1 and S_2 amounts to $\mathcal{S}_{\text{sos}}[\![S_1]\!] = \mathcal{S}_{\text{sos}}[\![S_2]\!]$. □

2.3 An equivalence result

We have given two definitions of the semantics of **While** and we shall now address the question of their equivalence.

Theorem 2.26 For every statement S of **While** we have $\mathcal{S}_{\text{ns}}[\![S]\!] = \mathcal{S}_{\text{sos}}[\![S]\!]$.

This result expresses two properties:

- If the execution of S from some state terminates in one of the semantics then it also terminates in the other and the resulting states will be equal.
- If the execution of S from some state loops in one of the semantics then it will also loop in the other.

It should be fairly obvious that the first property follows from the theorem because there are no stuck configurations in the structural operational semantics of **While**. For the other property suppose that the execution of S on state s loops in one of the semantics. If it terminates in the other semantics we have a contradiction with the first property because both semantics are deterministic (Theorem 2.9 and Exercise 2.22). Hence S will have to loop on state s also in the other semantics.

The theorem is proved in two stages as expressed by Lemma 2.27 and Lemma 2.28 below. We shall first prove:

Lemma 2.27 For every statement S of **While** and states s and s' we have

$$\langle S,\, s\rangle \to s' \text{ implies } \langle S,\, s\rangle \Rightarrow^* s'.$$

So if the execution of S from s terminates in the natural semantics then it will terminate in the same state in the structural operational semantics.

Proof: The proof proceeds by induction on the shape of the derivation tree for $\langle S,\, s\rangle \to s'$.

The case [ass$_{\text{ns}}$]: We assume that

$$\langle x := a,\, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$$

From [ass$_{\text{sos}}$] we get the required

$$\langle x := a,\, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

The case [skip$_{\text{ns}}$]: Analogous.

The case [comp$_{\text{ns}}$]: Assume that

$$\langle S_1;S_2,\ s\rangle \to s''$$

because

$$\langle S_1,\ s\rangle \to s' \text{ and } \langle S_2,\ s'\rangle \to s''$$

The induction hypothesis can be applied to both of the premises $\langle S_1,\ s\rangle \to s'$ and $\langle S_2,\ s'\rangle \to s''$ and gives

$$\langle S_1,\ s\rangle \Rightarrow^* s' \text{ and } \langle S_2,\ s'\rangle \Rightarrow^* s''$$

From Exercise 2.21 we get

$$\langle S_1;S_2,\ s\rangle \Rightarrow^* \langle S_2,\ s'\rangle$$

and thereby $\langle S_1;S_2,\ s\rangle \Rightarrow^* s''$.

The case [if$_{\text{ns}}^{\text{tt}}$]: Assume that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'$$

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \langle S_1,\ s\rangle \to s'$$

Since $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ we get

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle \Rightarrow^* s'$$

where the first relationship comes from [if$_{\text{sos}}^{\text{tt}}$] and the second from the induction hypothesis applied to the premise $\langle S_1,\ s\rangle \to s'$.

The case [if$_{\text{ns}}^{\text{ff}}$]: Analogous.

The case [while$_{\text{ns}}^{\text{tt}}$]: Assume that

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$$

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt},\ \langle S,\ s\rangle \to s' \text{ and } \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''$$

The induction hypothesis can be applied to both of the premises $\langle S,\ s\rangle \to s'$ and $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''$ and gives

$$\langle S,\ s\rangle \Rightarrow^* s' \text{ and } \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \Rightarrow^* s''$$

Using Exercise 2.21 we get

$$\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow^* s''$$

Using [while$_{\text{sos}}$] and [if$_{\text{sos}}^{\text{tt}}$] (with $\mathcal{B}[\![b]\!]s = \mathbf{tt}$) we get the first two steps of

$$\begin{aligned}&\langle \texttt{while } b \texttt{ do } S,\ s\rangle\\ &\qquad\Rightarrow \langle \texttt{if } b \texttt{ then } (S;\ \texttt{while } b \texttt{ do } S) \texttt{ else skip},\ s\rangle\\ &\qquad\Rightarrow \langle S;\ \texttt{while } b \texttt{ do } S,\ s\rangle\\ &\qquad\Rightarrow^* s''\end{aligned}$$

and we have already argued for the last part.

The case [while$^{\text{ff}}_{\text{ns}}$]: Straightforward. □

This completes the proof of Lemma 2.27. The second part of the theorem follows from:

Lemma 2.28 For every statement S of **While**, states s and s' and natural number k we have that

$$\langle S,\ s\rangle \Rightarrow^{\text{k}} s' \text{ implies } \langle S,\ s\rangle \to s'.$$

So if the execution of S from s terminates in the structural operational semantics then it will terminate in the same state in the natural semantics.

Proof: The proof proceeds by induction on the length of the derivation sequence $\langle S,\ s\rangle \Rightarrow^{\text{k}} s'$, that is by induction on k.

If k=0 then the result holds vacuously.

To prove the induction step we assume that the lemma holds for $\text{k} \le \text{k}_0$ and we shall then prove that it holds for k_0+1. We proceed by cases on how the first step of $\langle S,\ s\rangle \Rightarrow^{\text{k}_0+1} s'$ is obtained, that is by inspecting the derivation tree for the first step of computation in the structural operational semantics.

The case [ass$_{\text{sos}}$]: Straightforward (and $\text{k}_0 = 0$).

The case [skip$_{\text{sos}}$]: Straightforward (and $\text{k}_0 = 0$).

The cases [comp$^1_{\text{sos}}$] and [comp$^2_{\text{sos}}$]: In both cases we assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow^{\text{k}_0+1} s''$$

We can now apply Lemma 2.19 and get that there exists a state s' and natural numbers k_1 and k_2 such that

$$\langle S_1,\ s\rangle \Rightarrow^{\text{k}_1} s' \text{ and } \langle S_2,\ s'\rangle \Rightarrow^{\text{k}_2} s''$$

where $\text{k}_1+\text{k}_2=\text{k}_0+1$. The induction hypothesis can now be applied to each of these derivation sequences because $\text{k}_1 \le \text{k}_0$ and $\text{k}_2 \le \text{k}_0$. So we get

$$\langle S_1,\ s\rangle \to s' \text{ and } \langle S_2,\ s'\rangle \to s''$$

Using [comp$_{\text{ns}}$] we now get the required $\langle S_1;S_2,\ s\rangle \to s''$.

The case [if$^{\text{tt}}_{\text{sos}}$]: Assume that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle \Rightarrow^{\text{k}_0} s'$$

The induction hypothesis can be applied to the derivation sequence $\langle S_1,\ s\rangle \Rightarrow^{\text{k}_0} s'$ and gives

$$\langle S_1,\ s\rangle \to s'$$

The result now follows using [if$^{\text{tt}}_{\text{ns}}$].

The case [if$^{\text{ff}}_{\text{sos}}$]: Analogous.

The case [while$_{\text{sos}}$]: We have

$$\begin{aligned}&\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle\\ &\qquad \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle\\ &\qquad \Rightarrow^{\text{k}_0} s''\end{aligned}$$

The induction hypothesis can be applied to the k$_0$ last steps of the derivation sequence and gives

$$\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle \to s''$$

and from Lemma 2.5 we get the required

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s'' \qquad \square$$

Proof of Theorem 2.26: For an arbitrary statement S and state s it follows from Lemmas 2.27 and 2.28 that if $\mathcal{S}_{\text{ns}}[\![S]\!]s = s'$ then $\mathcal{S}_{\text{sos}}[\![S]\!]s = s'$ and vice versa. This suffices for showing that the functions $\mathcal{S}_{\text{ns}}[\![S]\!]$ and $\mathcal{S}_{\text{sos}}[\![S]\!]$ must be equal: if one is defined on a state s then so is the other, and therefore, if one is not defined on a state s then neither is the other. □

Exercise 2.29 Consider the extension of the language **While** with the statement `repeat` S `until` b. The natural semantics of the construct was considered in Exercise 2.7 and the structural operational semantics in Exercise 2.17. Modify the proof of Theorem 2.26 so that the theorem applies to the extended language. □

Exercise 2.30 Consider the extension of the language **While** with the statement `for` $x := a_1$ `to` a_2 `do` S. The natural semantics of the construct was considered in Exercise 2.8 and the structural operational semantics in Exercise 2.18. Modify the proof of Theorem 2.26 so that the theorem applies to the extended language. □

The proof technique employed in the proof of Theorem 2.26 may be summarized as follows:

Proof Summary for While:

Equivalence of two Operational Semantics

1: Prove by *induction on the shape of derivation trees* that for each derivation tree in the natural semantics there is a corresponding finite derivation sequence in the structural operational semantics.

2: Prove by *induction on the length of derivation sequences* that for each finite derivation sequence in the structural operational semantics there is a corresponding derivation tree in the natural semantics.

When proving the equivalence of two operational semantics for a language with additional programming constructs one may need to amend the above proof technique. One reason is that the equivalence result may have to be expressed differently from that of Theorem 2.26 (as will be the case if the extended language is non-deterministic). Also one might want to consider only some of the finite derivation sequences, for example those ending in a terminal configuration.

2.4 Extensions of While

In order to illustrate the power and weakness of the two approaches to operational semantics we shall consider various extensions of the language **While**. For each extension we shall show how to modify the operational semantics.

Abortion

We first extend **While** with the simple statement `abort`. The idea is that `abort` *stops* the execution of the complete program. This means that `abort` behaves differently from `while true do skip` in that it causes the execution to stop rather than loop. Also `abort` behaves differently from `skip` because a statement following `abort` will never be executed whereas one following `skip` certainly will.

Formally, the new syntax of statements is given by:

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\ & \mid & \texttt{while } b \texttt{ do } S \mid \texttt{abort} \end{array}$$

We shall not repeat the definitions of the sets of configurations but tacitly assume that they are modified so as to correspond to the extended syntax. The task that remains, therefore, is to define the new transition relations $\rightarrow$ and $\Rightarrow$.

The fact that `abort` stops the execution of the program is modelled by ensuring that the configurations of the form $\langle$`abort`, $s\rangle$ are *stuck*. Therefore the *natural semantics* of the extended language is still defined by the transition relation $\rightarrow$ of Table 2.1. So although the language and thereby the set of configurations have been extended we do not modify the definition of the transition relation. Similarly, the *structural operational semantics* of the extended language is still defined by Table 2.2.

From the structural operational semantics point of view it is clear now that `abort` and `skip` cannot be semantically equivalent. This is because

$$\langle \texttt{skip}, s\rangle \Rightarrow s$$

is the only derivation sequence for `skip` starting in s and

$$\langle \texttt{abort}, s\rangle$$

is the only derivation sequence for `abort` starting in s. Similarly, `abort` cannot be semantically equivalent to `while true do skip` because

$$\begin{aligned}
&\langle \texttt{while true do skip}, s\rangle\\
&\qquad\Rightarrow \langle \texttt{if true then (skip; while true do skip) else skip}, s\rangle\\
&\qquad\Rightarrow \langle \texttt{skip; while true do skip}, s\rangle\\
&\qquad\Rightarrow \langle \texttt{while true do skip}, s\rangle\\
&\qquad\Rightarrow \cdots
\end{aligned}$$

is an infinite derivation sequence for `while true do skip` whereas `abort` has none. Thus we shall claim that the structural operational semantics captures the informal explanation given earlier.

From the natural semantics point of view it is also clear that `skip` and `abort` cannot be semantically equivalent. However, it turns out that `while true do skip` and `abort` *are* semantically equivalent! The reason is that in the natural semantics we are only concerned with executions that terminate properly. So if we do not have a derivation tree for $\langle S, s\rangle \rightarrow s'$ then we cannot tell whether it is because we entered a stuck configuration or a looping execution. We can summarize this as follows:

Natural Semantics versus Structural Operational Semantics

- In a natural semantics we cannot distinguish between *looping* and *abnormal termination*.
- In a structural operational semantics *looping* is reflected by infinite derivation sequences and *abnormal termination* by finite derivation sequences ending in a stuck configuration.

We should note, however, that if abnormal termination is modelled by "normal termination" in a special error configuration (included in the set of terminal configurations) then we can distinguish between the three statements in both semantic styles.

Exercise 2.31 Theorem 2.26 expresses that the natural semantics and the structural operational semantics of **While** are equivalent. Discuss whether or not a similar result holds for **While** extended with `abort`. □

Exercise 2.32 Extend **While** with the statement

$\texttt{assert}\ b\ \texttt{before}\ S$

The idea is that if b evaluates to true then we execute S and otherwise the execution of the complete program aborts. Extend the structural operational semantics of Table 2.2 to express this (without assuming that **While** contains the `abort`-statement). Show that `assert true before` S is semantically equivalent to S but that `assert false before` S neither is equivalent to `while true do skip` nor `skip`. □

Non-determinism

The second extension of **While** has statements given by

$$\begin{array}{rcl} S & ::= & x := a \mid \texttt{skip} \mid S_1\ ;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid S_1\ \texttt{or}\ S_2 \end{array}$$

The idea is here that in S_1 `or` S_2 we can non-deterministically choose to execute either S_1 or S_2. So we shall expect that execution of the statement

$\texttt{x} := 1\ \texttt{or}\ (\texttt{x} := 2;\ \texttt{x} := \texttt{x} + 2)$

could result in a state where `x` has the value **1**, but it could as well result in a state where `x` has the value **4**.

When specifying the *natural semantics* we extend Table 2.1 with the two rules:

$$[\text{or}^1_{\text{ns}}] \qquad \frac{\langle S_1,\ s\rangle \to s'}{\langle S_1\ \texttt{or}\ S_2,\ s\rangle \to s'}$$

$$[\text{or}^2_{\text{ns}}] \qquad \frac{\langle S_2,\ s\rangle \to s'}{\langle S_1\ \texttt{or}\ S_2,\ s\rangle \to s'}$$

Corresponding to the configuration $\langle \texttt{x} := 1\ \texttt{or}\ (\texttt{x} := 2;\ \texttt{x} := \texttt{x}{+}2),\ s\rangle$ we have derivation trees for

$$\langle \texttt{x} := 1\ \texttt{or}\ (\texttt{x} := 2;\ \texttt{x} := \texttt{x}{+}2),\ s\rangle \to s[\texttt{x}{\mapsto}\mathbf{1}]$$

as well as

$$\langle \texttt{x := 1 or (x := 2; x := x+2)},\ s\rangle \to s[\texttt{x}\mapsto\mathbf{4}]$$

It is important to note that if we replace `x := 1` by `while true do skip` in the above statement then we will only have one derivation tree, namely that for

$$\langle \texttt{(while true do skip) or (x := 2; x := x+2)},\ s\rangle \to s[\texttt{x}\mapsto\mathbf{4}]$$

Turning to the *structural operational semantics* we shall extend Table 2.2 with the two axioms:

$$[\text{or}^1_{\text{sos}}] \qquad \langle S_1 \ \texttt{or}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$$

$$[\text{or}^2_{\text{sos}}] \qquad \langle S_1 \ \texttt{or}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$$

For the statement `x := 1 or (x := 2; x := x+2)` we have two derivation sequences:

$$\langle \texttt{x := 1 or (x := 2; x := x+2)},\ s\rangle \Rightarrow^* s[\texttt{x}\mapsto\mathbf{1}]$$

and

$$\langle \texttt{x := 1 or (x := 2; x := x+2)},\ s\rangle \Rightarrow^* s[\texttt{x}\mapsto\mathbf{4}]$$

If we replace `x := 1` by `while true do skip` in the above statement then we still have two derivation sequences. One is infinite

$$\begin{aligned}
&\langle \texttt{(while true do skip) or (x := 2; x := x+2)},\ s\rangle\\
&\qquad\Rightarrow\ \langle \texttt{while true do skip},\ s\rangle\\
&\qquad\Rightarrow^3 \langle \texttt{while true do skip},\ s\rangle\\
&\qquad\Rightarrow\ \cdots
\end{aligned}$$

and the other is finite

$$\langle \texttt{(while true do skip) or (x := 2; x := x+2)},\ s\rangle \Rightarrow^* s[\texttt{x}\mapsto\mathbf{4}]$$

Comparing the natural semantics and the structural operational semantics we see that the latter can choose the "wrong" branch of the `or`-statement whereas the first always chooses the "right" branch. This is summarized as follows:

Natural Semantics versus Structural Operational Semantics
• In a natural semantics *non-determinism will suppress looping*, if possible. • In a structural operational semantics *non-determinism does not suppress looping*.

Exercise 2.33 Consider the statement

$$\texttt{x := −1; while x≤0 do (x := x−1 or x := (−1)⋆x)}$$

Given a state s describe the set of final states that may result according to the natural semantics. Further describe the set of derivation sequences that are specified by the structural operational semantics. Based on this discuss whether or not you would regard the natural semantics as being equivalent to the structural operational semantics for this particular statement. □

Exercise 2.34 We shall now extend **While** with the statement

$$\texttt{random}(x)$$

and the idea is that its execution will change the value of x to be any positive natural number. Extend the natural semantics as well as the structural operational semantics to express this. Discuss whether $\texttt{random}(x)$ is a superfluous construct in the case where **While** is also extended with the `or` construct. □

Parallelism

We shall now consider an extension of **While** with a parallel construct. So now the syntax of expressions is given by

$$\begin{array}{rcl} S & ::= & x := a \mid \texttt{skip} \mid S_1\,;\,S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid S_1\ \texttt{par}\ S_2 \end{array}$$

The idea is that both statements of S_1 `par` S_2 have to be executed but that the execution can be *interleaved*. This means that a statement like

$$\texttt{x := 1 par (x := 2; x := x+2)}$$

can give three different results for `x`, namely **4**, **1** and **3**: If we first execute `x := 1` and then `x := 2; x := x+2` we get the final value **4**. Alternatively, if we first execute `x := 2; x := x+2` and then `x := 1` we get the final value **1**. Finally, we have the possibility of first executing `x := 2`, then `x := 1` and lastly `x := x+2` and we then get the final value **3**.

To express this in the *structural operational semantics* we extend Table 2.2 with the following rules:

$$[\text{par}^1_{\text{sos}}] \qquad \frac{\langle S_1,\, s\rangle \Rightarrow \langle S_1',\, s'\rangle}{\langle S_1\ \texttt{par}\ S_2,\, s\rangle \Rightarrow \langle S_1'\ \texttt{par}\ S_2,\, s'\rangle}$$

$$[\text{par}^2_{\text{sos}}] \qquad \frac{\langle S_1,\, s\rangle \Rightarrow s'}{\langle S_1\ \texttt{par}\ S_2,\, s\rangle \Rightarrow \langle S_2,\, s'\rangle}$$

$$[\text{par}^3_{\text{sos}}] \qquad \frac{\langle S_2,\ s\rangle \Rightarrow \langle S'_2,\ s'\rangle}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \Rightarrow \langle S_1\ \texttt{par}\ S'_2,\ s'\rangle}$$

$$[\text{par}^4_{\text{sos}}] \qquad \frac{\langle S_2,\ s\rangle \Rightarrow s'}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s'\rangle}$$

The first two rules take account of the case where we begin by executing the first step of statement S_1. If the execution of S_1 is not fully completed we modify the configuration so as to remember how far we have reached. Otherwise only S_2 has to be executed and we update the configuration accordingly. The last two rules are similar but for the case where we begin by executing the first step of S_2.

Using these rules we get the following derivation sequences for the example statement:

$$\begin{aligned}\langle \texttt{x := 1 par (x := 2; x := x+2)},\ s\rangle &\Rightarrow \langle \texttt{x := 2; x := x+2},\ s[\texttt{x}\mapsto\mathbf{1}]\rangle\\ &\Rightarrow \langle \texttt{x := x+2},\ s[\texttt{x}\mapsto\mathbf{2}]\rangle\\ &\Rightarrow s[\texttt{x}\mapsto\mathbf{4}]\end{aligned}$$

$$\begin{aligned}\langle \texttt{x := 1 par (x := 2; x := x+2)},\ s\rangle &\Rightarrow \langle \texttt{x := 1 par x := x+2},\ s[\texttt{x}\mapsto\mathbf{2}]\rangle\\ &\Rightarrow \langle \texttt{x := 1},\ s[\texttt{x}\mapsto\mathbf{4}]\rangle\\ &\Rightarrow s[\texttt{x}\mapsto\mathbf{1}]\end{aligned}$$

and

$$\begin{aligned}\langle \texttt{x := 1 par (x := 2; x := x+2)},\ s\rangle &\Rightarrow \langle \texttt{x := 1 par x := x+2},\ s[\texttt{x}\mapsto\mathbf{2}]\rangle\\ &\Rightarrow \langle \texttt{x := x+2},\ s[\texttt{x}\mapsto\mathbf{1}]\rangle\\ &\Rightarrow s[\texttt{x}\mapsto\mathbf{3}]\end{aligned}$$

Turning to the *natural semantics* we might start by extending Table 2.1 with the two rules:

$$\frac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \to s''}$$

$$\frac{\langle S_2,\ s\rangle \to s',\ \langle S_1,\ s'\rangle \to s''}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \to s''}$$

However, it is easy to see that this will not do because the rules only express that either S_1 is executed before S_2 or vice versa. This means that we have lost the ability to *interleave* the execution of two statements. Furthermore, it seems impossible to be able to express this in the natural semantics because we consider the execution of a statement as an atomic entity that cannot be split into smaller pieces. This may be summarized as follows:

Natural Semantics versus Structural Operational Semantics
• In a natural semantics the execution of the immediate constituents is an *atomic entity* so we cannot express interleaving of computations.   • In a structural operational semantics we concentrate on the *small steps* of the computation so we can easily express interleaving.

Exercise 2.35 Consider an extension of **While** that in addition to the par-construct also contains the construct

```
protect S end
```

The idea is that the statement S has to be executed as an atomic entity so that for example

```
x := 1 par protect (x := 2; x := x+2) end
```

only has two possible outcomes namely **1** and **4**. Extend the structural operational semantics to express this. Can you specify a natural semantics for the extended language? $\square$

Exercise 2.36 Specify a structural operational semantics for arithmetic expressions where the individual parts of an expression may be computed in parallel. Try to prove that you still obtain the result that was specified by $\mathcal{A}$. $\square$

2.5 Blocks and procedures

We now extend the language **While** with blocks containing declarations of variables and procedures. In doing so we introduce a couple of important concepts:

- variable and procedure environments, and
- locations and stores.

We shall concentrate on the natural semantics and will consider dynamic as well as static scope and non-recursive as well as recursive procedures.

Blocks and simple declarations

We first extend the language **While** with blocks containing declarations of local variables. The new language is called **Block** and its syntax is

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S_1 \,;\, S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D_V\ S\ \texttt{end} \end{array}$$

where D_V is a meta-variable ranging over the syntactic category $\mathbf{Dec}_V$ of *variable declarations*. The syntax of variable declarations is given by:

$$D_V \quad ::= \quad \texttt{var}\ x := a;\ D_V \mid \varepsilon$$

where ε is the empty declaration. The idea is that the variables declared inside a block begin D_V S end are *local* to it. So in a statement like

```
begin var y := 1;
        (x := 1;
        begin var x := 2; y := x+1 end;
        x := y+x)
end
```

the x in y := x+1 relates to the local variable x introduced by var x := 2, whereas the x in x := y+x relates to the global variable x that is also used in the statement x := 1. In both cases the y refers to the y declared in the outer block. Therefore, the statement y := x+1 assigns y the value **3**, rather than **2**, and the statement x := y+x assigns x the value **4**, rather than **5**.

Before going into the details of how to specify the semantics we shall define the set $\mathrm{DV}(D_V)$ of variables declared in D_V:

$$\begin{array}{lll} \mathrm{DV}(\texttt{var}\ x := a;\ D_V) & = & \{x\} \cup \mathrm{DV}(D_V) \\ \mathrm{DV}(\varepsilon) & = & \emptyset \end{array}$$

We next define the *natural semantics*. The idea will be to have one transition system for *each* of the syntactic categories **Stm** and $\mathbf{Dec}_V$. For statements the transition system is as in Table 2.1 but extended with the rule of Table 2.3. The transition system for variable declarations has configurations of the two forms $\langle D_V, s\rangle$ and s and the idea is that the transition relation $\rightarrow_D$ specifies the relationship between initial and final states as before:

$$\langle D_V, s\rangle \rightarrow_D s'$$

The relation $\rightarrow_D$ for variable declarations is given in Table 2.4. We generalize the substitution operation on states and write $s'[X{\longmapsto}s]$ for the state that is as s' except for variables in the set X where it is as specified by s. Formally,

[block$_{\text{ns}}$]	$\dfrac{\langle D_V,\, s\rangle \to_D s',\ \langle S,\, s'\rangle \to s''}{\langle \texttt{begin}\ D_V\ S\ \texttt{end},\, s\rangle \to s''[\text{DV}(D_V)\longmapsto s]}$

Table 2.3: Natural semantics for statements of **Block**

[var$_{\text{ns}}$]	$\dfrac{\langle D_V,\, s[x\mapsto\mathcal{A}[\![a]\!]s]\rangle \to_D s'}{\langle \texttt{var}\ x := a;\ D_V,\, s\rangle \to_D s'}$
[none$_{\text{ns}}$]	$\langle \varepsilon,\, s\rangle \to_D s$

Table 2.4: Natural semantics for variable declarations

$$(s'[X\longmapsto s])\ x = \begin{cases} s\ x & \text{if } x \in X \\ s'\ x & \text{if } x \notin X \end{cases}$$

This operation will ensure that local variables are restored to their previous values when the block is left.

Exercise 2.37 Use the natural semantics of Table 2.3 to show that execution of the statement

```
begin var y := 1;
        (x := 1;
        begin var x := 2; y := x+1 end;
        x := y+x)
end
```

will lead to a state where x has the value **4**. □

It is somewhat harder to specify a *structural operational semantics* for the extended language. One approach is to replace states with a structure that is similar to the run-time stacks used in the implementation of block structured languages. Another is to extend the statements with fragments of the state. However, we shall not go further into this.

Procedures

We shall now extend the language **Block** with procedure declarations. The syntax of the language **Proc** is:

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1\ ;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D_V\ D_P\ S\ \texttt{end} \mid \texttt{call}\ p \\ D_V & ::= & \texttt{var}\ x := a;\ D_V \mid \varepsilon \\ D_P & ::= & \texttt{proc}\ p\ \texttt{is}\ S;\ D_P \mid \varepsilon \end{array}$$

Here p is a meta-variable ranging over the syntactic category **Pname** of procedure names; in the concrete syntax there need not be any difference between procedure names and variable names but in the abstract syntax it is convenient to be able to distinguish between the two. Furthermore, D_P is a meta-variable ranging over the syntactic category $\mathbf{Dec}_P$ of *procedure declarations*.

We shall give three different semantics of this language. They differ in their choice of scope rules for variables and procedures:

- dynamic scope for variables as well as procedures,
- dynamic scope for variables but static scope for procedures, and
- static scope for variables as well as procedures.

To illustrate the difference consider the statement

```
begin var x := 0;
          proc p is x := x ⋆ 2;
          proc q is call p;
          begin var x := 5;
                    proc p is x := x + 1;
                    call q; y := x
          end
end
```

If *dynamic scope* is used for variables as well as procedures then the final value of y is **6**. The reason is that `call q` will call the *local* procedure p which will update the *local* variable x. If we use dynamic scope for variables but *static scope* for procedures then y gets the value **10**. The reason is that now `call q` will call the *global* procedure p and it will update the *local* variable x. Finally, if we use static scope for variables as well as procedures then y gets the value **5**. The reason is that `call q` now will call the *global* procedure p which will update the *global* variable x so the local variable x is unchanged.

Dynamic scope rules for variables and procedures

The general idea is that to execute the statement `call` p we shall execute the body of the procedure. This means that we have to keep track of the association of procedure names with procedure bodies. To facilitate this we shall introduce the notion of a *procedure environment*. Given a procedure name the procedure environment env_P will return the statement that is its body. So env_P is an element of

[ass$_{\text{ns}}$]	$env_P \vdash \langle x := a,\, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$
[skip$_{\text{ns}}$]	$env_P \vdash \langle \texttt{skip},\, s\rangle \to s$
[comp$_{\text{ns}}$]	$\dfrac{env_P \vdash \langle S_1,\, s\rangle \to s',\; env_P \vdash \langle S_2,\, s'\rangle \to s''}{env_P \vdash \langle S_1;S_2,\, s\rangle \to s''}$
[if$^{\text{tt}}_{\text{ns}}$]	$\dfrac{env_P \vdash \langle S_1,\, s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
[if$^{\text{ff}}_{\text{ns}}$]	$\dfrac{env_P \vdash \langle S_2,\, s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
[while$^{\text{tt}}_{\text{ns}}$]	$\dfrac{env_P \vdash \langle S,\, s\rangle \to s',\; env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\, s'\rangle \to s''}{env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\, s\rangle \to s''}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
[while$^{\text{ff}}_{\text{ns}}$]	$env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\, s\rangle \to s$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
[block$_{\text{ns}}$]	$\dfrac{\langle D_V,\, s\rangle \to_D s',\; \text{upd}_{\text{P}}(D_P,\, env_P) \vdash \langle S,\, s'\rangle \to s''}{env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\, s\rangle \to s''[\text{DV}(D_V) \longmapsto s]}$
[call$^{\text{rec}}_{\text{ns}}$]	$\dfrac{env_P \vdash \langle S,\, s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\, s\rangle \to s'}$ where $env_P\ p = S$

Table 2.5: Natural semantics for **Proc** with dynamic scope rules

$$\mathbf{Env}_{\text{P}} = \mathbf{Pname} \hookrightarrow \mathbf{Stm}$$

The next step will be to extend the natural semantics to take the environment into account. We shall extend the transition system for statements to have transitions of the form

$$env_P \vdash \langle S,\, s\rangle \to s'$$

The presence of the environment means that we can always access it and therefore get hold of the bodies of declared procedures. The result of modifying Table 2.1 to incorporate this extra information is shown in Table 2.5.

Concerning the rule for begin D_V D_P S end the idea is that we update the procedure environment so that the procedures declared in D_P will be available when executing S. Given a global environment env_P and a declaration D_P, the updated procedure environment, $\text{upd}_\text{P}(D_P, env_P)$, is specified by:

$$\text{upd}_\text{P}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_P) = \text{upd}_\text{P}(D_P,\ env_P[p \mapsto S])$$

$$\text{upd}_\text{P}(\varepsilon,\ env_P) = env_P$$

As the variable declarations do not need to access the procedure environment it is not necessary to extend the transition system for declarations with the extra component. So for variable declarations we still have transitions of the form

$$\langle D,\ s\rangle \to_D s'$$

The relation is defined as for the language **Block**, that is by Table 2.4.

We can now complete the specification of the semantics of blocks and procedure calls. Note that in the rule [block_ns] of Table 2.5 we use the updated environment when executing the body of the block. In the rule [$\text{call}_\text{ns}^\text{rec}$] for procedure calls we make use of the information provided by the environment. It follows that procedures will *always* be recursive.

Exercise 2.38 Consider the following statement of **Proc**:

```
begin proc fac is begin var z := x;
                        if x = 1 then skip
                        else (x := x−1; call fac; y := z⋆y)
                  end;
      (y := 1; call fac)
end
```

Construct a derivation tree for the execution of this statement from a state s where $s\ \texttt{x} = \mathbf{3}$. □

Exercise 2.39 Use the semantics to verify that the statement

```
begin var x := 0;
      proc p is x := x ⋆ 2;
      proc q is call p;
      begin var x := 5;
            proc p is x := x + 1;
            call q; y := x
```

[call$_{\text{ns}}$]	$$\frac{env'_P \vdash \langle S,\, s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\, s\rangle \to s'}$$ where $env_P\ p = (S,\, env'_P)$
[call$^{\text{rec}}_{\text{ns}}$]	$$\frac{env'_P[p{\mapsto}(S,\, env'_P)] \vdash \langle S,\, s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\, s\rangle \to s'}$$ where $env_P\ p = (S,\, env'_P)$

Table 2.6: Procedure calls in case of mixed scope rules (choose one)

```
        end
    end
```

considered earlier does indeed assign the expected value to `y`. □

Static scope rules for procedures

We shall now modify the semantics of **Proc** to specify static scope rules for procedures. The first step will be to extend the procedure environment env_P so that procedure names are associated with their body as well as the procedure environment at the point of declaration. To this end we define

$$\mathbf{Env}_{\mathrm{P}} = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_{\mathrm{P}}$$

This definition may seem problematic because $\mathbf{Env}_{\mathrm{P}}$ is defined in terms of itself. However, this is not really a problem because a concrete procedure environment always will be built from smaller environments starting with the empty procedure environment. The function $\mathrm{upd}_{\mathrm{P}}$ updating the procedure environment can then be redefined as:

$$\mathrm{upd}_{\mathrm{P}}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_P) = \mathrm{upd}_{\mathrm{P}}(D_P,\ env_P[p{\mapsto}(S,\ env_P)])$$
$$\mathrm{upd}_{\mathrm{P}}(\varepsilon,\ env_P) = env_P$$

The semantics of variable declarations are unaffected and so is the semantics of most of the statements. Compared with Table 2.5 we shall only need to modify the rules for procedure calls. In the case where the procedures of **Proc** are assumed to be *non-recursive* we simply consult the procedure environment to determine the body of the procedure and the environment at the point of declaration. This is expressed by the rule [call$_{\text{ns}}$] of Table 2.6. In the case where the procedures of **Proc** are assumed to be *recursive* we have to make sure that occurrences of `call` p inside the body of p refer to the procedure itself. We shall therefore update the procedure environment to contain that information. This is expressed by the rule

[$\text{call}_{\text{ns}}^{\text{rec}}$] of Table 2.6. The remaining axioms and rules are as in Tables 2.5 (without [$\text{call}_{\text{ns}}^{\text{rec}}$]) and 2.4. (Clearly a choice should be made between [call_{ns}] or [$\text{call}_{\text{ns}}^{\text{rec}}$].)

Exercise 2.40 Construct a statement that illustrates the difference between the two rules for procedure call given in Table 2.6. Validate your claim by constructing derivation trees for the executions of the statement from a suitable state. □

Exercise 2.41 Use the semantics to verify that the statement of Exercise 2.39 assigns the expected value to y. □

Static scope rules for variables

We shall now modify the semantics of **Proc** to specify static scope rules for variables as well as procedures. To achieve this we shall replace the states with two mappings: a *variable environment* that associates a *location* with each variable and a *store* that associates a value with each location. Formally, we define a variable environment env_V as an element of

$$\textbf{Env}_\textbf{V} = \textbf{Var} \to \textbf{Loc}$$

where **Loc** is a set of locations. For the sake of simplicity we shall take **Loc** = **Z**. A store *sto* is an element of

$$\textbf{Store} = \textbf{Loc} \cup \{ \text{ next } \} \to \textbf{Z}$$

where 'next' is a special token used to hold the next free location. We shall need a function

$$\text{new: } \textbf{Loc} \to \textbf{Loc}$$

that given a location will produce the next one. In our case where **Loc** is **Z** we take 'new' to be the successor function on the integers.

So rather than having a single mapping s from variables to values we have split it into two mappings env_V and *sto* and the idea is that $s = sto \circ env_V$. To determine the value of a variable x we shall first

- determine the location $l = env_V\ x$ associated with x and then
- determine the value $sto\ l$ associated with the location l.

Similarly, to assign a value v to a variable x we shall first

- determine the location $l = env_V\ x$ associated with x and then
- update the store to have $sto\ l = v$.

$[\text{var}_{\text{ns}}]$	$\dfrac{\langle D_V,\ env_V[x\mapsto l],\ sto[l\mapsto v][\text{next}\mapsto\text{new } l]\rangle \to_D (env'_V,\ sto')}{\langle \texttt{var}\ x := a;\ D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')}$ where $v = \mathcal{A}[\![a]\!](sto\circ env_V)$ and $l = sto$ next
$[\text{none}_{\text{ns}}]$	$\langle \varepsilon,\ env_V,\ sto\rangle \to_D (env_V,\ sto)$

Table 2.7: Natural semantics for variable declarations using locations

The initial variable environment could for example map all variables to the location **0** and the initial store could for example map 'next' to **1**. The variable environment (and the store) is updated by the variable declarations. The transition system for variable declarations is therefore modified to have the form

$$\langle D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')$$

because a variable declaration will modify the variable environment as well as the store. The relation is defined in Table 2.7. Note that we use '*sto* next' to determine the location l to be associated with x in the variable environment. Also the store is updated to hold the correct value for l as well as 'next'. Finally note that the declared variables will get positive locations.

To obtain static scoping for variables we shall extend the procedure environment to hold the variable environment at the point of declaration. Therefore env_P will now be an element of

$$\textbf{Env}_{\text{P}} = \textbf{Pname} \hookrightarrow \textbf{Stm} \times \textbf{Env}_{\text{V}} \times \textbf{Env}_{\text{P}}$$

The procedure environment is updated by the procedure declarations as before, the only difference being that the current variable environment is supplied as an additional parameter. The function upd_{P} is now defined by:

$$\begin{aligned}
&\text{upd}_{\text{P}}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_V,\ env_P) = \\
&\qquad \text{upd}_{\text{P}}(D_P,\ env_V,\ env_P[p\mapsto(S,\ env_V,\ env_P)]) \\
&\text{upd}_{\text{P}}(\varepsilon,\ env_V,\ env_P) = env_P
\end{aligned}$$

Finally, the transition system for statements will have the form:

$$env_V,\ env_P \vdash \langle S,\ sto\rangle \to sto'$$

so given a variable environment and a procedure environment we get a relationship between an initial store and a final store. The modification of Tables 2.5 and 2.6 is rather straightforward and is given in Table 2.8. Note that in the new rule for blocks there is no analogue of $s''[\text{DV}(D_V)\longmapsto s]$ as the values of variables only can be obtained by accessing the environment.

$[\mathrm{ass_{ns}}]$	$\mathit{env}_V,\ \mathit{env}_P \vdash \langle x := a,\ \mathit{sto}\rangle \to \mathit{sto}[l{\mapsto}v]$ where $l = \mathit{env}_V\ x$ and $v = \mathcal{A}[\![a]\!](\mathit{sto}{\circ}\mathit{env}_V)$
$[\mathrm{skip_{ns}}]$	$\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{skip},\ \mathit{sto}\rangle \to \mathit{sto}$
$[\mathrm{comp_{ns}}]$	$\dfrac{\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_1,\ \mathit{sto}\rangle \to \mathit{sto}',\ \ \mathit{env}_V,\ \mathit{env}_P \vdash \langle S_2,\ \mathit{sto}'\rangle \to \mathit{sto}''}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_1;S_2,\ \mathit{sto}\rangle \to \mathit{sto}''}$
$[\mathrm{if_{ns}^{tt}}]$	$\dfrac{\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_1,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ \mathit{sto}\rangle \to \mathit{sto}'}$ if $\mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{tt}$
$[\mathrm{if_{ns}^{ff}}]$	$\dfrac{\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_2,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\ \mathit{sto}\rangle \to \mathit{sto}'}$ if $\mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{ff}$
$[\mathrm{while_{ns}^{tt}}]$	$\dfrac{\begin{array}{c}\mathit{env}_V,\ \mathit{env}_P \vdash \langle S,\ \mathit{sto}\rangle \to \mathit{sto}',\\ \mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ \mathit{sto}'\rangle \to \mathit{sto}''\end{array}}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ \mathit{sto}\rangle \to \mathit{sto}''}$ if $\mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{tt}$
$[\mathrm{while_{ns}^{ff}}]$	$\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{while}\ b\ \mathtt{do}\ S,\ \mathit{sto}\rangle \to \mathit{sto}$ if $\mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{ff}$
$[\mathrm{block_{ns}}]$	$\dfrac{\begin{array}{c}\langle D_V,\ \mathit{env}_V,\ \mathit{sto}\rangle \to_D (\mathit{env}'_V,\ \mathit{sto}'),\\ \mathit{env}'_V,\ \mathit{env}'_P \vdash \langle S,\ \mathit{sto}'\rangle \to \mathit{sto}''\end{array}}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{begin}\ D_V\ D_P\ S\ \mathtt{end},\ \mathit{sto}\rangle \to \mathit{sto}''}$ where $\mathit{env}'_P = \mathrm{upd_P}(D_P,\ \mathit{env}'_V,\ \mathit{env}_P)$
$[\mathrm{call_{ns}}]$	$\dfrac{\mathit{env}'_V,\ \mathit{env}'_P \vdash \langle S,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{call}\ p,\ \mathit{sto}\rangle \to \mathit{sto}'}$ where $\mathit{env}_P\ p = (S,\ \mathit{env}'_V,\ \mathit{env}'_P)$
$[\mathrm{call_{ns}^{rec}}]$	$\dfrac{\mathit{env}'_V,\ \mathit{env}'_P[p{\mapsto}(S,\ \mathit{env}'_V,\ \mathit{env}'_P)] \vdash \langle S,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \mathtt{call}\ p,\ \mathit{sto}\rangle \to \mathit{sto}'}$ where $\mathit{env}_P\ p = (S,\ \mathit{env}'_V,\ \mathit{env}'_P)$

Table 2.8: Natural semantics for **Proc** with static scope rules

Exercise 2.42 Apply the natural semantics of Table 2.8 to the factorial statement of Exercise 2.38 and a store where the location for x has the value **3**. □

Exercise 2.43 Verify that the semantics applied to the statement of Exercise 2.39 gives the expected result. □

Exercise 2.44 * An alternative semantics of the language **While** is defined by the axioms and rules [ass_{ns}], [skip_{ns}], [comp_{ns}], [$\text{if}^{\text{tt}}_{\text{ns}}$], [$\text{if}^{\text{ff}}_{\text{ns}}$], [$\text{while}^{\text{tt}}_{\text{ns}}$] and [$\text{while}^{\text{ff}}_{\text{ns}}$] of Table 2.8. Formulate and prove the equivalence between this semantics and that of Table 2.1. □

Exercise 2.45 Modify the syntax of procedure declarations so that procedures take two *call-by-value* parameters:

$$D_P ::= \texttt{proc}\ p(x_1,x_2)\ \texttt{is}\ S;\ D_P \mid \varepsilon$$

$$S ::= \cdots \mid \texttt{call}\ p(a_1,a_2)$$

Procedure environments will now be elements of

$$\mathbf{Env_P} = \mathbf{Pname} \hookrightarrow \mathbf{Var} \times \mathbf{Var} \times \mathbf{Stm} \times \mathbf{Env_V} \times \mathbf{Env_P}$$

Modify the semantics given above to handle this language. In particular, provide new rules for procedure calls: one for non-recursive procedures and another for recursive procedures. Construct statements that illustrate how the new rules are used. □

Exercise 2.46 Now consider the language **Proc** and the task of achieving *mutual recursion*. The procedure environment is now defined to be an element of

$$\mathbf{Env_P} = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env_V} \times \mathbf{Env_P} \times \mathbf{Dec_P}$$

The idea is that if $env_P\ p = (S,\ env'_V,\ env'_P,\ D^\star_P)$ then $D^\star_P$ contains all the procedure declarations that are made in the same block as p. Define upd'_P by

$$\text{upd}'_P(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_V,\ env_P,\ D^\star_P) =$$
$$\qquad \text{upd}'_P(D_P,\ env_V,\ env_P[p{\mapsto}(S,\ env_V,\ env_P,D^\star_P)],\ D^\star_P)$$

$$\text{upd}'_P(\varepsilon,\ env_V,\ env_P,D^\star_P) = env_P$$

Next redefine upd_P by

$$\text{upd}_P(D_P,\ env_V,\ env_P) = \text{upd}'_P(D_P,\ env_V,\ env_P,\ D_P)$$

Modify the semantics of **Proc** so as to obtain mutual recursion among procedures defined in the same block. Illustrate how the new rules are used on an interesting statement of your choice.

(Hint: Convince yourself, that [$\text{call}^{\text{rec}}_{\text{ns}}$] is the only rule that needs to be changed; then consider whether or not the function upd_P might be useful in the new definition of [$\text{call}^{\text{rec}}_{\text{ns}}$].) □

Exercise 2.47 We shall consider a variant of the semantics where we use the variable environment rather than the store to hold the next free location. So assume that

$$\mathbf{Env_V} = \mathbf{Var} \cup \{ \text{ next } \} \to \mathbf{Loc}$$

and

$$\mathbf{Store} = \mathbf{Loc} \to \mathbf{Z}$$

As before we shall write $sto \circ env_V$ for the state obtained by first using env_V to find the location of the variable and then sto to find the value of the location. The clauses of Table 2.7 are now replaced by

$$\frac{\langle D_V,\ env_V[x{\mapsto}l][\text{next}{\mapsto}\text{new } l],\ sto[l{\mapsto}v]\rangle \to_D (env'_V,\ sto')}{\langle \texttt{var}\ x := a;\ D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')}$$
$$\text{where } v = \mathcal{A}[\![a]\!](sto{\circ}env_V) \text{ and } l = env_V \text{ next}$$

$$\langle \varepsilon,\ env_V,\ sto\rangle \to_D (env_V,\ sto)$$

Construct a statement that computes different results under the two variants of the semantics. Validate your claim by constructing derivation trees for the executions of the statement from a suitable state. □