UNIVERSITY OF ADELAIDE · LIBRARY ·
9.9.92

ATLANTIS: A TOOL FOR LANGUAGE DEFINITION AND INTERPRETER SYNTHESIS

Michael John Oudshoorn, B.Sc.(Hons.)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ADELAIDE

August 1992

Abstract

Programming language semantics are usually defined informally in some form of technical natural language, or in a very mathematical manner with techniques such as the Vienna Definition Method (VDM) or denotational semantics. One difficulty which arises from serious attempts to define language semantics is that the resulting definition is generally suitable for a single limited kind of reader. For example, the more formal kind of definition may suit a compiler writer or a language designer, but will be less convenient for other potential classes of reader, such as programmers. The latter frequently make use of some completely separate description (e.g., an introductory text book on the language); not surprisingly, inconsistencies between these separate descriptions and the language definition are commonplace.

This thesis develops a technique for the definition of programming language semantics which is suitable for a wide range of potential readers. This technique employs an operational semantic model which is based on the algebraic specification of abstract data types; the semantic model manipulates multi-layer descriptions of language semantics and supports multiple passes in these descriptions.

The semantic technique described in this thesis lends itself to the semi-automatic generation of an interpreter from the language definition, a fact which acts as an incentive to language designers to produce a formal definition of any new programming language, since the prototype implementation allows experimentation with new language features and their semantics. The system which generates an interpretive implementation from a language definition is called ATLANTIS, A Tool for LANguage definiTion and Interpreter Synthesis, and is also described in this thesis.

Declaration

This is to certify that this thesis contains no material which has previously been accepted for the award of any degree or diploma in any University. To the best of my knowledge and belief it contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

If this thesis is accepted for the award of the degree, permission is granted for it to be made available for loan and photocopying.

Michael Oudshoorn

August 1992

Acknowledgments

I wish to acknowledge all the help and encouragement that I have received throughout my candidature as a Ph.D. student. In particular, I am indebted to my supervisor, Prof. Chris Marlin, for his advice, thoughts, guidance and encouragement; and to Dr. David Bover who acted as my supervisor whilst Prof. Marlin was on study leave during the second half of 1988 and early 1989. I also wish to express my thanks to Prof. Peter King and Dr. Brian Molinari for their comments on this thesis. I am also indebted to my wife, Jo, and my son, Daniel, for having to tolerate many late nights, and often having to come second behind my study. Many thanks to my parents, Bep and Nick, who supported and encouraged my study.

I also wish to thank the other members of the staff, the postgraduate students and visiting speakers, who have all contributed to this thesis in many small ways. Of particular note in this regard are Michael McCarthy and Ali Mili. I also wish to acknowledge the contribution of Keith Ransom, who developed the Ada code in Section 5.5 and carried out the experiments referred to in Section 5.7.

I would also like to thank all those companies that allowed me time on their machines and gave me access to their Pascal compilers to obtain the results discussed in Chapter 1; I have been asked by many of these companies not to publish their names.

Contents

List of Tables

List of Figures

UNIVERSITY OF ADELAIDE 9/9/93 LIBRARY

Chapter 1

Introduction

With a myriad of computer programming languages in use on a great many computers, it has become particularly important to be able to specify the syntax and semantics of each language in some fashion. These specifications, called *language definitions*, must be precise in order to allow a programmer to use the same language on different machines with confidence and consistent results. Hence, a language specification helps to make software written in the specified language portable to another computer, even though the machine architecture and operating system may be different, without the need for in any laborious work to convert the program into a form suitable for the target machine. Programmers are clear beneficiaries of such language definitions.

Compiler writers and language designers also use these language definitions. To the compiler writer, the language definition represents the specification of the end-product to be attained – the language implementation. To the language designer, the language definition is the vehicle by which the language concepts and details are conveyed to those interested; the process of designing a new language may also involve examining a number of existing language definitions, in order to make comparisons.

It is clear that the language definition is of interest to many classes of user, and is thus destined to serve a multitude of users with different expectations, desires and needs. Generally, there is only one definitive document which constitutes the "language definition" and this document must serve the diverse needs of these various users.

It is possible to identify two aspects of a language definitions: a description of the syntactic elements of the language, and a discussion of its semantic aspects. Each of these aspects can be described either informally or formally, as discussed in the next two sections.

1.1 Descriptions of Syntax

The syntactic description of a language can be described informally through the use of examples. This approach is not common in a language definition, nor is it acceptable. It leaves open the possibility of not covering all syntactic aspects adequately, and of leaving the reader confused and unsure.

A common method of describing the syntax of a language, which is relatively precise, is that of syntax charts or railroad diagrams. This technique tends to be very easy to read and understand, but consumes relatively large amounts of space in a document. The graphical nature of the charts makes them useful to programmers, but makes them awkward for compiler writers and language designers. Care must be taken when producing a syntax chart to ensure that only valid language constructs are defined. It is all too easy to merge together seemingly identical parts of the syntax chart to inadvertently produce a syntax chart that admits illegal syntax.

Another common approach for the description of syntax is to use Backus-Naur Form (BNF) [100], or Extended Backus-Naur Form (EBNF) [158]. These are relatively formal and precise techniques which are well understood, as well as being easy to produce and read. These techniques cover the syntactic structure of the language and are more useful to a reader than the informal approach as a reference for answering questions about the syntax of a language, since the answer can be obtained directly from the BNF or EBNF definition rather than deduced or guessed from a set of examples.

1.2 Descriptions of Semantics

1.2.1 The Informal Approach

Semantic descriptions for languages are also available in two forms: informal and formal. One informal approach is to describe the language semantics by examples accompanied with a brief natural language narrative. This approach appears to have been adopted, for example, in the definition of PS-algol [97]. Such an approach is ideal for learning a new language, but it is of little use to implementors and more advanced users of the language asking technical questions regarding the semantics of various constructs.

The most common, and widely accepted, method for describing the semantics of a programming language is a natural language approach. Several examples of the application of this method exist. Some employ a relatively informal natural language narrative in order to describe the semantics of all the various aspects of the programming language; such a description is reasonably easy to read. Examples include Pascal's early definitions [65, 155], the definition of Modula-2 [159] and that for Oberon [160]. This style of definition is acceptable to programmers wishing to learn a new language, and to more experienced programmers with language queries. However, it is not so useful to compiler writers, as evidenced by the fact the many early Pascal compilers were inconsistent with regard to the semantics of a number of language constructs [85, 152]. The difficulty with these natural language descriptions is that they allow multiple correct interpretations, hence leading to the observed inconsistencies between various compilers.

In order to satisfy the need for an internally consistent document and a consistent interpretation of this document, the current trend in the natural language definition of the programming language semantics employs many technical terms. As inconsistencies between the language definition and implementation were uncovered with the early definitions, further technical terms were introduced and the style of writing became more technical and precise, to the point that the language definitions written in natural language are more akin to legal documents than useful, easy to read and understand descriptions. This change in emphasis was necessary in order to reduce the possibility of misunderstanding and ambiguity. The result is that the language definitions are more useful to compiler writers and language designers, but less useful to programmers. Examples of such descriptions include the current definitions of Pascal [17], Extended Pascal [162], ANSI C [161], Mesa [95] and Ada [141]. Due to the very nature of a natural language description, there are always misunderstandings and ambiguities, simply because different people may interpret the same wording slightly differently and hence derive a slightly different meaning from a sentence. These misunderstandings manifest themselves in the form of incorrect compilers which do not strictly adhere to the specifications intended by the language designer and are, perhaps, inconsistent with other compilers.

Semantic definitions in terms of natural language are not only difficult to write and understand, but they are often difficult to use as a source from which to learn the language. As a result, books on specific programming languages are written in less technical terms, once again introducing further scope for misunderstanding and ambiguity. These books, although easier to read, usually do not contain all the information found in the language definition; typical examples include text books for the programming language Ada (such as [9, 14, 39, 149]) which do not aim to discuss the entire language, but rather help programmers read the language and use it. These books generally serve to spread the author's interpretation of the contents of the language definition rather than the language designer's intentions. Text books also have a very different goal to that of the language definition. Text books aim to introduce a new language to the readers and are concerned with teaching the basics so that the new language can be used, whilst a language definition is not concerned with teaching, but is rather focussed on precise definition.

Programmers often realize that text books do not form a suitable source from which to answer language questions. Rather than turning to a text book which merely provides someone else's interpretation of the language standard, programmers often write a few test programs to examine their behaviour with respect to one or more implementations of the language. This latter approach amounts to making use of a language implementation as a language definition, and it should be remembered that each implementation only represents a particular interpretation of the actual language definition, namely that of the particular implementation team. The implementation is perhaps as likely to suffer difficulties as the text book.

Precise natural language definitions of programming language semantics are difficult to produce and minor variations in the description of an aspect of the language may have far-reaching consequences. This is examined in Section 1.2.1.1, where some historical changes to the Pascal language definition are considered. However, not only is it difficult to maintain consistency between the various revisions of a language definition, but it is also difficult to ensure that all aspects of the language are adequately covered. Section 1.2.1.2 examines the potential difficulties encountered as a result of a simple query about an aspect of a programming language and demonstrates that compiler writers may also have difficulty with the language definition.

1.2.1.1 Historical Changes to the Pascal Definition

Natural language definitions of programming languages are extremely difficult to write in a precise fashion. For example, consider the attempts to express the fact that the loop control variable in a "**for**" statement in Pascal may not be explicitly modified. Over the history of Pascal thus far, there have been several attempts to describe this intention, and these are summarized by Cooper in [24]. In the original description of Pascal [155], Wirth states:

> "The repeated statement S must alter neither the value of the control variable nor the final value."

This definition did not preclude the statement S from invoking a procedure which modified the value of the control variable. As a result, this rule was later altered somewhat in Jensen and Wirth [65] to specify that the expressions representing the initial and final values of the control variable be evaluated only once:

> "The control variable, the initial value, and the final value must be of the same scalar type (or subrange thereof), and must not be altered by the repeated statement."

The first British Standards Institution (BSI) draft of the Pascal standard [16] broadened the restriction on the control variable even further:

> "An error is caused if the control variable is assigned to by the repeated statement or altered by any procedure or function activated by the repeated statement."

The word "error" has roughly the same meaning in [16] as it does in the current definition of Pascal [17], where it is defined as "a violation by a program of the requirements of this standard that a processor is permitted to leave undetected". The first International Organization for Standardization (ISO) draft of the Pascal standard [64] tightened up the limitations on the control variable by saying:

> "Assigning references to the control variable shall not occur with the repeated statement."

The definition of "assigning reference" was such that any change in the control variable was virtually precluded (but data flow analysis would be required to detect all such changes). The second draft of the ISO standard [63] had a slightly reworded definition of the same restriction.

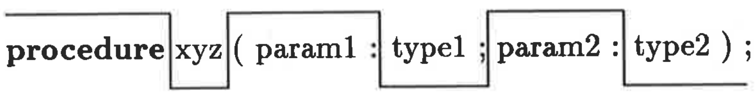
As can be seen from the above example, the standard has undergone several changes in the words used in attempts to describe the same idea. With each change of words came a slightly different semantics for the language Pascal. Due to the difficulties of implementing a compiler that detects all such changes to control variables whilst still running at reasonable speeds, the restrictions have been relaxed again in the current standard [17], where only certain cases are mentioned and discussed as being "threatening" (having the potential to alter the value of the control variable).

The above brief examination of the history of the question of modifications to a loop control variable over the lifetime of the programming language Pascal is sufficient to illustrate the increasing complexity and increasing amounts of technical jargon that have found their way into the definition of this language. A major difficulty with these technical terms is that each language definition tends to use terms peculiar to the language being described, sometimes using the same term found in another language definition but with a different meaning. Although the definitions of these terms are to be found somewhere in the relevant language definition, they are often scattered throughout the language definition and are hence extremely difficult to look up. This is made even more difficult when the technical terms do not appear in the index to the definition.

1.2.1.2 Problems within the Current Pascal Definition

The effectiveness of any language definition technique may be gauged by how easily a member of the intended target audience can use the definition to answer a question regarding the language semantics. The intended users of a natural language definition of a programming language such as Pascal includes programmers and compiler writers.

One could easily imagine a scenario where several programmers were discussing a bug in some code. During the discussion, several comments and suggestions may be made regarding the action required to discover and correct the bug. Some of suggestions may involve the alteration of code already written, others may include the addition or removal of code, and there may be questions regarding the semantics of aspects of the language. Imagine that the language in question is Pascal; in this case, one such question on language semantics may be related to the scope of identifiers in a parameter list. An initial hypothesis may be that the formal parameter identifiers belong to the scope of the procedure or function to which this is a parameter list, and that the type identifiers are interpreted in the scope of the surrounding block:



In order to decide whether or not this is the case, two approaches may be followed. A typical programmer's approach may well be to develop a small suite of test programs to test the hypothesis, and the typical compiler writer's approach is to examine the language standard. Both approaches will now be examined in turn.

Two test programs that a programmer may write are given in Figure 1.1. The programmer would then compile the programs and draw conclusions regarding the language semantics based on the outcome of these compilations. This approach suffers a major drawback in that the compiler is being used as the language definition, rather than the compiler being regarded as the compiler writer's attempt to model the language semantics. This approach can be pursued further by compiling these small test programs on a variety of machines under a variety of operating systems, in an effort to discard incorrect interpretations. This was done and showed that, of the twelve Pascal systems used, eleven detected an error at the identifier "a" marked by "{*}" in the program "test1" in Figure 1.1(a). The error related to the fact that "a" was no longer a type identifier at this point. One compiler detected no errors at all. For program "test2" in Figure 1.1(b), eleven compilers detected no errors at all, but one compiler indicated that the identifier "a" at "{*}" was redefined after use "in this scope". Only one compiler detected an error in both programs, and one detected no errors in either.

```
program test1;
   type
      a = char;
      b = boolean;
   procedure x1(a:b; c:a{*});
      begin {x1}
      end; {x1}
   begin {test1}
   end. {test1}
```

(a)

```
program test2;
   type
      a = char;
      b = boolean;
   procedure x2(c:a{#}; a{*}:b);
      begin {x2}
      end; {x2}
   begin {test2}
   end. {test2}
```

(b)

Figure 1.1. Two of the test programs developed.

These results would indicate that the initial hypothesis was incorrect, for if it were correct then the use of the identifier "a" as both a parameter and a type would have been accepted. The problem now is to find what is the correct interpretation of the results. Of the twelve compilers used, ten had identical results, one complained of no errors, one of an error in each. It is possible that these latter two compilers are incorrect and their results should be discarded.

Of the ten remaining compilers, it would appear that the order of the declarations inside the parameter list was important. However, this is contrary to intuition, since the order of declaration of variables is immaterial at any other point in the program. At this point, several questions may be asked about the semantics of Pascal's parameter lists, such as "Is the order of declaration important?", "What is the scope of each identifier in the parameter list?" and "Is a parameter list like '(a:a)' valid?". The simple test programs used have not satisfactorily confirmed the original hypothesis, but instead have served to confuse the issue even further and generate many more questions. In fact, the tests have served to give the programmer an incorrect understanding of the language definition. The only option left now is to consult the standard and see what can be learned from it, forcing the programmer to follow the same route as the compiler writer. However, it is interesting to note that few programmers are likely to test their code as thoroughly as this. Most will simply write a single test program, compile it on a single machine under a single operating system, and trust the results obtained. Clearly, this is a dangerous assumption.

The alternative approach to writing test programs is to consult the language definition on the matter concerned. On this occasion, the hypothesis concerns the scope of identifiers within a parameter list. The initial starting point would be to check the section of the language standard dealing with parameters; this is Section 6.6.3 of [17]. The section itself is divided into several subsections which are entitled: "General", "Value parameters", "Variable parameters", "Procedural parameters", "Functional parameters", "Parameter list congruity", "Conformant array parameters" and "Conformability". From this list, a guess must be made as to which subsection to read first. The sections on value and variable parameters prove to be no use because the discussion there is limited to the relationship between the actual and formal parameters. This only leaves the section labelled "General" (Section 6.6.3.1) as being potentially helpful. Part of this section reads:

> 6.6.3.1 **General** ...
>
> The occurrence of an identifier in the identifier-list of a value-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal parameter.

NOTE 2. If the formal-parameter-list is contained in a procedural-parameter-specification or a functional-parameter-specification, there is no corresponding procedure-block or function-block.

Here, the reader is presented with much technical natural language jargon. Unless already familiar with the standard, the reader must now look up the meanings of several words such as defining-point and value-parameter-specification, and find out precisely what is meant by these. Once this has been done, the reader realizes that this section of the standard does little to answer the question because the scope of the type identifiers is not discussed.

At this point, types or scope may be looked up in the language definition. Looking up information regarding types proves to be fruitless. Section 6.2.2 of [17] relates to scope and it contains the following subsections:

6.2.2.7 When an identifier or label has a defining-point for a region, another identifier or label with the same spelling shall not have a defining-point for that region.

6.2.2.8 Within the scope of a defining-point of an identifier or label, each occurrence of an identifier or label having the same spelling as the identifier or label of the defining-point shall be designated an applied occurrence of the identifier or label of the defining-point, except for an occurrence that constituted the defining-point of that identifier or label; such an occurrence shall be designated a defining occurrence. No occurrence outside that scope shall be an applied occurrence.

NOTE. Within the scope of a defining-point of an identifier or label, there

> are no occurrences of an identifier or label that cannot be distinguished from it and have a defining-point for a region enclosing that scope.
>
> 6.2.2.9 The defining-point of an identifier or label shall precede all applied occurrences of that identifier or label contained by the program-block with one exception, namely that an identifier may have an applied occurrence in the type-identifier of the domain-type of any new-pointer-types contained by the type-definition-part that contains the defining-point of the type-identifier.

These three subsections finally give the answer to the question. Both programs are in error because the defining-point of each parameter extends over the entire formal-parameter-list and all other uses of that identifier are an applied occurrence of that identifier.

Program "test1" of Figure 1.1(a) is in error at the point labelled "{∗}" because the identifier "a" at this point is an applied occurrence of the formal parameter "a" defined previously within the formal-parameter-list and is illegal because a type-identifier is expected. All except one of the twelve compilers detected this error correctly. Program "test2" in Figure 1.1(b) is also in error, but at the point labelled "{#}". The identifier declared at "{∗}" is a defining-point for the identifier "a" for the entire region that is the formal-parameter-list, and so the identifier "a" at {#} is an applied occurrence. By 6.2.2.8 of [17], this is an error and should be flagged accordingly. Only one compiler of the twelve found anything wrong with program "test2" and it flagged the error at {∗} for the wrong (albeit related) reason. The result is that eleven of the twelve Pascal compilers used did not conform to the standard. This clearly illustrates how dangerous it is to use a compiler as a convenient form of the language definition when testing some hypothesis about the semantics of the language.

A small test suite of twenty Pascal programs similar to those in Figure 1.1 was run under each of the twelve Pascal implementations. The result was astounding. Not one implementation correctly handled all of the test programs. The most likely conclusion to be drawn from these results is that many of the compiler writers were having difficulty understanding the standard or were misunderstanding the technical terms in the natural language description which constitutes the language definition.

1.2.2 Formal Definitions

Programming language definitions written in natural language are moderately useful to the programming community, but are far from the perfect solution to the problem of describing the semantics of a programming language to that community. Many people have recognized the deficiency of the natural language approach and, as a direct result, formal specifications of some languages have been developed (such as for Pascal [68, 148] and Ada [62, 140]). Most formal descriptions of programming languages to date are not themselves the language definition, but are based on the informal natural language language definition, hence admitting the possibility of ambiguity.

One popular formal method of describing the semantics of a programming language is the attribute grammar technique. This is more mathematical and precise than natural language and is more useful to compiler writers and language designers than it is to programmers wishing to answer a simple query on the semantics of the language. Attribute grammar descriptions are more difficult to produce and difficult to read than a natural language description for any realistic programming language, such as Pascal [68] and Ada [140]. However, attribute grammars avoid many of the problems that are inherent in natural language definitions, due to their formal approach. They are well suited to describing the static context-dependent properties of a programming language, but they are not quite so useful in describing the dynamic semantics [119]. For example, the attribute grammar description of Pascal presented in Kastens *et al.* [68] does not fully describe the semantics of a Pascal "**for**" statement, since the final value of the loop control variable is not mentioned (although it is discussed in the relevant language standard).

Not only is an attribute grammar description of a language difficult to produce, but the descriptions tend to be rather lengthy. Despite this, however, benefit may be obtained from taking the natural language definition of a language and producing an attribute grammar description. Watt [148], for example, produced an attribute grammar description of the Jensen and Wirth definition of Pascal [65]. Even though this description was incomplete and really only formalized the concept of type compatibility in Pascal, several problems with the informal Jensen and Wirth definition were uncovered.

Attribute grammars have proved useful in the automatic generation of components of compilers. The GAG system [68] has been used to produce front-end components for compilers for Pascal [68] and Ada [140]. This capability alone gives attribute grammars an advantage over natural language descriptions, as it demonstrates that they provide more than mere language documentation.

Another popular formalism for the definition of programming language semantics is the denotational approach [2, 46, 127, 134]. This approach has been successfully used to define the semantics (or aspects of the semantics) of several programming languages (e.g., [3, 23, 37, 90, 101, 120, 138]). In addition, the technique has been successfully applied to the comparison of the semantics of two languages whose syntax and approach to programming differ somewhat [91]. The technique itself, however, is very mathematical and is difficult to produce and read. Its mathematical basis tends to suggest that it is amenable to the automatic generation of aspects of an implementation. The major drawback to the latter is that the denotational approach draws on such a rich mathematical domain that automatic generation of an implementation prototype from a denotational description must often place restrictions on the mathematical concepts which may be employed [113, 114].

The denotational approach appears to be principally aimed at the definition of programming language semantics, and less so toward the formal generation of a compiler, or interpreter, based on the formal description. Typically, approaches based on the denotational semantic approach only generate one aspect of an implementation. In order to gain a complete implementation, the approach must be combined with other tools which usually take different forms of input. This makes the approach less attractive to a language designer who wishes to generate a prototype implementation directly from s single language definition.

The language standard for Modula-2 [156, 157, 159] currently being developed by the BSI utilizes the Vienna Development Method (VDM) [13]. This standardization process [15, 151] will result in a document that precisely describes the semantics of Modula-2; along with its accompanying natural language description, it will provide a definition that is usable by the general community. Unfortunately, it is not yet possible to automatically generate a compiler from the VDM definition (although work is proceeding in this area), and those wishing to read a formal VDM definition will require a higher level of mathematical training than those reading a natural language definition. Another difficulty is that VDM has itself not been standardized, and was chosen in preference to an axiomatic approach because there was doubt that an axiomatic approach could handle a description of that size.

Various researchers have employed a multitude of formalisms to formally define a programming language with the express purpose of generating a compiler from the specification [133, 142, 148], or generating an environment from the specification [96, 118]. These approaches may require that machine code be generated, and hence a formal description of the target language may also be required; alternatively, some predefined intermediate code such as abstract syntax trees or denotational semantics (such as the intermediate representations used in [98, 99, 113, 114, 147]) may be produced, which is then translated into machine code. These attempts at formal language definition are oriented towards the automatic generation of a compiler, or aspects of it, and the formal definition of the programming language is not the principal concern. Typical examples include: yacc [66] which is aimed at parser generation; Linguist-86 [31] and GAG [68] which are aimed at front-end generation (syntactic and static semantic analysis); the work of Ganapathi and Fischer [35, 36], Bird [12], and Graham *et al.* [40, 47, 48] on code generator generators; and the work of Lee [76] on the translation phase of a compiler. All of this work is important, but in each case the definition of the programming language has been secondary. Further, to combine these tools to produce an entire language implementation involves multiple language definitions as the tools do not use a common input format.

Another formal specification technique, more typically applied to the specification of problems such as telephone networks [59], is the specification language Z [28]. Duke *et al.* [29] have used the technique to describe the static semantics of a small language. Z appears to be limited in that the dynamic semantics of a programming language cannot be adequately described using the technique. Further difficulties in using Z as a technique for the specification of programming language semantics arise since Z is continually changing, as new features as developed for use with Z specifications, in much the same way that VDM is an evolving specification technique.

Other formal techniques for the description of programming language semantics exist, but they currently lack the popularity of attribute grammars. Most of these methods, such as w-grammars [143, 144, 145], relator calculus [124, 125] and Hoare axiomatics [20, 57, 60], are also highly mathematical and equally difficult to use in producing aspects of a programming language definition. As with attribute grammars, they tend to be more useful to compiler writers and language designers than to computer programmers.

1.3 The Model Presented in this Thesis

The situation has arisen where two language descriptions are really required, since the needs of compiler writers and language designers are vastly different to those of the computer programmer. However, the use of two language descriptions is awkward and error-prone, as there are likely to be discrepancies and minor differences between them. These difficulties are having to be addressed by the BSI standardization effort of Modula-2 discussed above. The ideal solution is to have one document that defines the language clearly and formally, being useful at the same time to compiler writers, language designers and programmers. With such a simple language definition, there should be no ambiguities or misunderstandings. The technique should be such that the author of the definition should be forced to examine all areas of the language, giving due consideration to each aspect so that a complete language definition is produced.

This thesis develops and uses a model of programming language semantics which can satisfy the disparate needs of the various potential users of the definition. The model is an information structure model [150], meaning that the programming language semantics are defined in terms of manipulations of information structures. Such an approach has been successfully used to describe the semantics of various coroutine facilities in several languages [33, 83], to describe and compare interprocess communication mechanisms [84] and to develop a language definition for a new language [33, 83], but in each case the "primitive" operations of the model were defined informally in natural language. A similar model is used in this thesis, but it is extended to provide a formal basis. This basis is achieved through the slightly unconventional use of the algebraic specification technique for the definition of abstract data types (ADT's). Building on this formal base, the model is extended further to become a multi–layer model. Such a model is capable of dealing with both the static and dynamic semantics of programming languages.

By developing a model that consists of several layers, it is possible to have a single description that caters for the vastly different needs of various groups. Language designers and compiler writers can find the precise definitions that they require, whilst programmers and language enthusiasts can simply read to the level most convenient to them.

This model then forms the basis of a system called ATLANTIS, A Tool for LANguage definiTion and Interpreter Synthesis, which provides a mechanism of the formal definition of a programming language in terms of an operational semantic model as well as a means for the semiautomatic generation of an interpreter. The generated language implementation exactly mimics the building and manipulation of information structures and subsequently provides a reference implementation which is faithful to the language definition. This allows language designers quick access to a prototype implementation of a new language and hence they are able to experiment with and evaluate language concepts before finalizing the language definition.

The information structures used to define a language are likely to be similar, or identical, to those needed to describe similar languages. As a result, the information structures, and hence the central layer of the model, are reusable between language definitions; this reduces the amount of work needed to define a new language. Language comparison at a formal level is also facilitated in this way.

The reliance on an information structure model and the expression of programming language semantics through transformations on these information structures means that the technique described in this thesis is likely to be applicable to a wide range of language descriptions covering many different programming paradigms. In fact, any language whose semantics can be expressed via transformations on suitable information structures could be defined using the technique; the difficulty in applying it to a new language lies in the discovery of suitable information structures and ensuring that they can be defined using the algebraic specification technique for abstract data types. For the purpose of illustration, this thesis will concentrate on the definition of block structured sequential programming languages and illustrate how the model can be expanded to handle aspects of intertask communication in a parallel programming language.

The remainder of this thesis is organized as follows: Chapter 2 examines the algebraic specification technique for abstract data types; this technique is used as the basis of the information structure model proposed for the definition of the semantics of sequential programming languages. This model is described in detail in Chapter 3, including the multi-layer and multi-pass aspects of the model. This is followed, in

Chapter 4, by a description of the ATLANTIS system based on the model introduced in Chapter 3. Chapter 5 examines the special difficulties encountered when describing parallel languages and examines alterations to the ATLANTIS system to cater for this class of languages. The final chapter presents some conclusions and discusses possible future directions from the work described in this thesis.

Chapter 2

Algebraic Specification of Abstract Data Types

As indicated at the end of the previous chapter, a technique for language definition will be presented in this thesis. This technique is based on an information structure model of programming language semantics. If this model is to be used as the basis for a language definition, care must be taken to ensure that the information structure itself is described formally. The description of the information structure and associated operations has been a weakness with previous information structure models, in that these information structures have typically been defined informally (e.g., the models used in [26, 83, 93]).

The information structures underlying these models can, however, be regarded as abstract data types (ADT's). If this is done, then the information structures and their associated operations can be described in a precise fashion using algebraic specification techniques for the specification of ADT's. As a prelude to the use of these techniques in later chapters, this chapter presents a survey of relevant aspects of these algebraic specification techniques.

2.1 An Example of an Abstract Data Type

In order to illustrate the issues involved in specifying ADT's algebraically, this chapter will develop an ADT specification from the original conception of the ADT to its formal verification. The ADT which will be developed describes a table data structure. This data structure is common, fairly simple and likely to be already well understood by the reader. The table type will consist of an "index", that is a key, to indicate which item within the table is required and a description related to that index; the latter is referred to as the "information". The index must be unique and is used to identify which information object in the table is to be selected; each information object is associated with precisely one index. The table is similar to the kind of one-dimensional array which can be found in many existing programming languages. The table type places no restrictions on the index or information types, provided that the index values can be distinguished from each other and a test for equality is available for index values. The index and information types could be integers, enumerated types, or even other ADT's defined by the user. The index and information types may well be different.

The operations defined on the ADT describing the table type should be sufficient to enter, retrieve and manipulate the data within it; this ADT will be called "Table". A user will need to be able to define a new object of type Table and check to see if this object contains any index/information pairs. To have a useful data type, the user will need to be able to insert items (i.e., index/information pairs) into a Table object, remove such items from a Table object, and alter the information associated with a particular index. It will also be essential to be able to retrieve the information associated with an index already known inside the Table. The operations constitute the minimum set of operations to make effective use of an ADT representing a table.

Understanding the data structure is the first stage in developing an ADT. It is necessary to understand how the ADT is likely to be used and what operations are required to manipulate it effectively. Every conceivable operation need not be provided, as more complex operations can be built from a basic minimum set. Once the function of the ADT and the operations that are to be provided are understood, an axiomatic specification of the ADT can be built.

2.2 Axiomatizations

There are two alternative approaches to the formal definition of an ADT, namely the *initial* and *final* algebra approaches. In both cases, the data type is treated as an algebra, as described in [163, 164], and hence strict mathematical reasoning is allowed. With the initial algebra approach [43], any two objects from the type of interest are considered equal if and only if such equality follows directly from the axioms. This allows the development of equivalence classes of ground terms (terms without variables); for example,

$$\{\ 0, 1\text{–}1, 2\text{–}2, \ldots, 1 \times 0, 2 \times 0, \ldots\}$$

is an equivalence class for the constant "0". Each member of the equivalence class can be shown to be identical via the axioms defining the type integer and hence if one such member occurs in an equation, it may be replaced by any other member of the equivalence class without any loss of information or generality. The final algebra approach [51, 53, 55] takes the opposite viewpoint. Any two objects of the type of interest are considered equal as long as this view does not conflict with any of the axioms. Throughout this thesis, we will use the initial algebra approach, but attempt to use a specification technique similar to that presented in [51] and [55]. The reason for choosing the initial algebra approach was not because it is easier to understand or comprehend, but simply because a choice had to be made. Either approach would have been satisfactory for this application.

Recent developments in algebraic specifications [21] allow the automatic generation of templates from specifications; one example is the work on the ASSPEGIQUE system [10, 11, 21]. Work in this area is evolving and undergoing continuous improvement and modification. As a result, it has been decided to not incorporate these techniques into the ATLANTIS system, but rather leave the option of merging ATLANTIS with a system such as ASSPEGIQUE at a later date. This merger would produce a system capable of producing an entire implementation prototype from a formal specification of the language.

Following the kind of development in Section 2.1, we have only an informal description of the data type and such a description is unsuitable as the basis for an information structure model. In order to obtain a formal basis for a description of the data structure, it is necessary to consider each operation and define its effect on the data structure. This is done by producing a set of axioms that formally specify the behaviour of the operations. The data type is said to be *axiomatized* when such a set of axioms has been produced. Such an axiomatic description of a data type has well defined mathematical properties which can be used to ensure that the axiomatic description behaves as expected.

The axioms derived need to be as concise and as specific as possible in order to avoid any ambiguity. Confusion and ambiguity is often introduced by superfluous axioms and thus only axioms vital to the description of the data type should be presented. The set of axioms that finally describe the data type should be restrictive, so that no implementations which adhere to the axioms but represent a data type that differs from the one intended can be created. However, the axioms should not be overly restrictive, so as to preclude any valid implementation of the data type. In effect, the axiomatic description should describe the data type fully and in a manner that allows only one correct interpretation, but does not preclude any valid implementations, nor allow any invalid implementations.

The axioms need to satisfy two mathematical properties in order to be truly useful: *consistency* and *sufficient-completeness* [41, 43, 44, 50, 51]. The first of these means that the axioms must not produce a conflicting result under any circumstances. If two or more axioms are applicable, then they must all lead to an identical result. If this is not the case, then the axiom set is said to be *inconsistent.* The second property, sufficient-completeness, means that the axioms must describe all valid objects that belong to the ADT. Both properties will be discussed in more detail later.

Following the style of specification that is presented by Guttag in [51] and [55], an axiomatization of the ADT from Section 2.1 can be written as shown in Figure 2.1. The eleven axioms in Figure 2.1 formally describe this ADT. Each axiom says something of importance about the abstract data type. Axiom 1 indicates that the "empty" operation applied to a "new_table" will return the value "true", whilst Axiom 2 indicates that when the "empty" operation is applied to any object of type Table other than the "new_table" object, the value "false" will be returned.

Axioms 3 and 4 describe the action of the "member" operation. Axiom 3 states that when the "member" operation is applied to the "new_table", regardless of the value of the index supplied as the second argument, the value "false" is returned. In contrast to this, Axiom 4 indicates that the "member" operation returns "true" if "index_1" is identical to "index_2", where "index_1" was used in an "insert" operation which was the last operation applied to the table, and "index_2" is the second argument to

ADT Table [index,
information]
sorts Table/index, information, boolean

where index **has** equal: index × index → boolean

comments

The operator “new_table” creates an object of type Table.
The function “empty” tests to see if a Table object is equal to “new_table”.
The operator “member” indicates whether or not a particular item is an index into the given Table object; i.e., it tests to see if there is an item in the Table object with the given index value.
The operator “insert” will insert an item into a Table object, replacing any existing item which has that index value.
The operator “remove” deletes the item with the specified index value if it is present. Otherwise, an error occurs.
The operator “search” returns the information associated with an index value if that index value is present. Otherwise, an error occurs.
The operator “alter” changes the information associated with an item in the Table if the relevant index value is present. Otherwise, an error is issued.

syntax

new_table:		→	Table
empty:	Table	→	boolean
*member:	Table × index	→	boolean
insert:	Table × index × information	→	Table
remove:	Table × index	→	Table ⋃ {*error*}
search:	Table × index	→	information ⋃ {*error*}
alter:	Table × index × information	→	Table ⋃ {*error*}

semantics

declare tab: Table
index_1, index_2, index_3: index
info_1, info_2: information

axioms

(1) empty(new_table) = true
(2) empty(insert(tab, index_1, info_1)) = false
(3) member(new_table, index_1)) = false

Figure 2.1. Algebraic specification of the abstract data type “Table”.

```
(4)   member(insert(tab, index_1, info_1), index_2) =
          if equal(index_1, index_2)
          then
            true
          else
            member(tab, index_2)
          end if
(5)   insert(insert(tab, index_1, info_1), index_2, info_2) =
          if member(insert(tab, index_1, info_1), index_2)
          then
            alter(insert(tab, index_1, info_1), index_2, info_2)
          else
            insert(insert(tab, index_2, info_2), index_1, info_1)
          end if
(6)   remove(insert(tab, index_1, info_1), index_2) =
          if equal(index_1, index_2)
          then
            tab
          else
            insert(remove(tab, index_2), index_1, info_1)
          end if
(7)   remove(new_table, index_1) = error
(8)   search(insert(tab, index_1, info_1), index_2) =
          if equal(index_1, index_2)
          then
            info_1
          else
            search(tab, index_2)
          end if
(9)   search(new_table, index_1) = error
(10)  alter(tab, index_1, info_1) =
          insert(remove(tab, index_1), index_1, info_1)
(11)  alter(new_table, index_1, info_1) = error
```

Figure 2.1. Continued.

"member". If the indices differ, then the "member" operation is applied recursively to the Table with the last index/information pair removed (i.e., it is applied to the Table without the last "insert" being applied). If this removal results in the "new_table" object, then Axiom 3 becomes applicable and "false" is returned.

The description of the "insert" operation is presented by Axiom 5. It states that if an item that is already in a Table object is to be inserted then, rather than allowing two items with the same index value, the previously inserted item is altered to hold the information that is now being inserted. If the index value for the item whose insertion is now being attempted is not already present in the Table object, then the order in which the objects are inserted is not relevant. The data type described in Figure 2.1 will be known as an "Unordered_Table" because of this characteristic.

A formal description of the "remove" operation is given by Axioms 6 and 7. Axiom 6 states that a "remove" operation applied immediately after an "insert" operation results in the original Table object value if the indices in the "remove" and the "insert" operations are identical; that is, insertion and immediate removal of an item with a particular index will have no overall effect on the Table object. If the indices in the two operation applications differ, then the "remove" operation is applied to the Table object value which existed prior to the insertion taking place, and the "insert" operation is then applied to the resulting object. Axiom 7 indicates that the application of the "remove" operation to a "new_table" will result in an "*error*" value being returned. This is because the "remove" operation has no defined or sensible meaning when applied to a "new_table". To express this inapplicability, the error value is appended to the range of values which could result from the "remove" operation. This error value is treated as though it were a member of the range of certain ADT operations, whereas it is really a special value [55].

The "search" operation is described by Axioms 8 and 9. Axiom 8 states that if the "search" operation is applied to a Table object, then the last information value inserted is returned if the given index value is equal to the index value supplied in that last insertion. If the indices differ, then the "search" operation is applied recursively to the Table with the last item removed. If no item with index value "index_2" is present in the Table, an attempt will eventually be made to apply the "search" operation to the "new_table". This situation is covered by Axiom 9, which indicates that this results in an error value.

The operation "alter" is described as the equivalent of applying a "remove" operation followed by an "insert" operation, these operations being applied to items which both have the indicated index value. This is stated formally by Axiom 10. The result of applying the "alter" operation to a "new_table" is described by Axiom 11, which ensures an error value is returned in this case.

The axioms themselves do not constitute the entire ADT description. From Figure 2.1, it can be seen that the type name of the ADT ("Table" in this instance) is specified in the heading, along with the names of two types. The type names stand for any arbitrary types with which the ADT may be instantiated and they amount to a form of parameterization for the ADT (as discussed further in [6, 25, 38, 111, 115, 139], for example); their function is similar to that of a formal parameter in the parameter list to a procedure or function. The **sorts** clause indicates all the sorts (types) that the ADT definition employs; any sorts not listed in this clause may not be used within the ADT. The sorts listed before the "/" character are those that are defined by this ADT definition and those following the "/" character are those used by it. The **where** clause indicates any restrictions that these sorts must satisfy; in the case of the specification in Figure 2.1, the equality operator is assumed to exist for the index type. This operator is required in the axioms explained earlier. The **comments** part of the specification consist of informal comments to aid the reader of the specification. They have no place in a formal description, other than as minor explanatory notes. All details regarding the behaviour of the ADT must be obtainable from the axioms. The next section of the ADT specification defines the syntax of each of the operations; this is also known as the *signature* of the ADT. For example, the "member" operation takes an object of sort Table and an object of sort index, and returns a boolean value. Finally, the semantics of the ADT is defined in an axiomatic manner, as discussed earlier. All variables used in the axioms are first declared of the appropriate sort; they may then stand for any valid object of their specified sort.

The observant reader will have noticed that one operation not discussed in the informal description of the ADT has been added. The operation "member" was introduced into the algebraic specification, as it is necessary for the specification of Axiom 5. This new "member" operation must also be defined in the specification in a formal and rigorous manner. At this point, a decision must be made. The user can be given free access to this operation, or it can be hidden from the user but still used freely within the specification and implementation of the ADT. To indicate that the user has no access to a particular operation, it is common to prefix it with a "*" in the section describing the syntax.

2.3 Error Conditions

An error value was introduced into the specification in the previous section, in order to handle invalid terms. This error value is appended to the range of valid results allowed by some operations and is necessary in order to indicate erroneous conditions.

$$\begin{aligned}
\textit{error} &= \text{remove}(\textit{error}, \text{index\_1}) \\
&= \text{remove}(\text{insert}(\text{new\_table}, \text{index\_1}, \textit{error}), \text{index\_1}) \\
&= \text{remove}(\text{insert}(\text{new\_table}, \text{index\_1}, && \textit{by Axiom 9} \\
&\qquad\qquad \text{search}(\text{new\_table}, \text{index\_1})), \text{index\_1}) \\
&= \text{new\_table} && \textit{by Axiom 6}
\end{aligned}$$

Figure 2.2. Collapsing the abstract data type "Table".

As can be seen in Axioms 7, 9 and 11, it is possible for the user to attempt undesirable operations, such as trying to remove an item from an empty Table. If this is attempted, the error value is returned. In an implementation, this could result in an exception being raised and handled, or an error being reported in some way. These details are left to the implementor – it is enough to indicate in the specification that the error value is to be returned. It is assumed that each operation has an implicit axiom indicating that if an operation application includes the error value in its parameter list, then the error value is propagated as the result of the operation application. Hence, if an error value is returned during any part of the evaluation of an expression, the value of the expression is the error value.

Although this approach (which follows [51]) appears to be reasonable, it contains a hidden trap for the unwary. Following this approach, it is possible to "collapse" the entire ADT and demonstrate clearly undesirable results. One such result is shown in Figure 2.2, which uses the axioms of the specification in Figure 2.1 to show that the error value and "new_table" are equivalent. Since all objects are constructed by "insert" operations applied to "new_table", and since "insert" operations applied to an error value yield error values, then we really have a meaningless specification.

Thus, following the suggestion of Guttag on how to handle error conditions leads to the conclusion that all objects of the sort have the error value. This is known as the ***horror of collapsing types.*** The main cause for this is the manner in which *error* values are propagated. It is clearly not beneficial to presume the existence of implicit axioms that cause an operation to return the error value if any of the parameters to the operation is the error value.

A method to rectify this problem is suggested by Goguen *et al.* in [42]. This method adds the error value as a constant to each sort; for example, "E" may be used to represent the error value associated with the sort "boolean". Along with the introduction of such constants, it is also necessary to add some additional operations. As an introduction to the method suggested by Goguen *et al.*, the specification for the sort boolean will be presented.

In the specification for the sort boolean, it is not only necessary to add the constant "E" to denote the error value, but also to supply the unary operator "OK". This operator, when applied to a constant within the boolean sort, returns "true" if the value is either "true" or "false", and "false" if the value is "E". To avoid the horror of collapsing types, it is necessary to ensure that each variable in an axiom is not equivalent to the error value unless specifically stated.

The operator "OK" is defined on the constants belonging to the sort boolean as follows:

$$\begin{aligned}&\text{OK(true)} = \text{true}\\&\text{OK(false)} = \text{true}\\&\text{OK(E)} = \text{false}\end{aligned}$$

It is now necessary to define the behaviour of the predicate "OK" with respect to terms in boolean logic. This behaviour can be specified as follows:

$$\begin{aligned}&\text{OK}(\sim B_1) = \text{OK}(B_1)\\&\text{OK}(B_1 \wedge B_2) = \text{OK}(B_1) \wedge \text{OK}(B_2)\\&\text{OK}(B_1 \vee B_2) = \text{OK}(B_1) \vee \text{OK}(B_2)\end{aligned}$$

where "B_1" and "B_2" are values from the sort boolean. It is also necessary to give axioms stating the behaviour of operators when one of the values involved is the error value. Since it is required that error values propagate when they are detected, all operations involving the error value should return the error value as its result. Thus,

$$\begin{aligned}
&{\sim}\mathrm{E} = \mathrm{E}\\
&\mathrm{E} \wedge \mathrm{B} = \mathrm{E}\\
&\mathrm{B} \wedge \mathrm{E} = \mathrm{E}\\
&\mathrm{E} \vee \mathrm{B} = \mathrm{E}\\
&\mathrm{B} \vee \mathrm{E} = \mathrm{E}
\end{aligned}$$

where "B" is any value from the sort boolean, including the value "E". An extended conditional is now also needed. This conditional, "IFE", behaves in the same manner as a conventional if-then-else construct in the case of a valid condition. If the condition evaluates to the error value, then the error value is returned. Its behaviour is defined by:

$$\begin{aligned}
&\mathrm{IFE}(\mathrm{true}, \mathrm{B}_1, \mathrm{B}_2) = \mathrm{B}_1\\
&\mathrm{IFE}(\mathrm{false}, \mathrm{B}_1, \mathrm{B}_2) = \mathrm{B}_2\\
&\mathrm{IFE}(\mathrm{E}, \mathrm{B}_1, \mathrm{B}_2) = \mathrm{E}
\end{aligned}$$

where "B_1" and "B_2" are again values of the sort boolean. It is these operators, "OK" and "IFE", in conjunction with the error value, "E", which provide the basis for the improved handling of error values in the ADT specification. Since, in general, equations are likely to have some positive number, say n, of arguments, it is convenient to make use of a derived operator defined by:

$$\mathrm{OK}_n(\mathrm{B}_1, \ldots, \mathrm{B}_n) = \wedge_{i=1}^{n} \mathrm{OK}(\mathrm{B}_i)$$

where the "B_i" represent values from the sort boolean. To make this approach workable, a derived conditional is also introduced:

$$\mathrm{IFOK}_n(\mathrm{B}_1, \ldots, \mathrm{B}_n, \mathrm{B}) = \mathrm{IFE}(\mathrm{OK}_n(\mathrm{B}_1, \ldots, \mathrm{B}_n), \mathrm{B}, \mathrm{E})$$

where the "B_i" and "B" are values from the sort boolean. The value of the above expression is the value of "B" if $OK(B_i)$ has the value "true", for all i such that $1 \leq i \leq n$, and has the value "E" otherwise.

Now an axiomatic description of the type boolean can be presented, as shown in Figure 2.3, without fear of the specification collapsing into something meaningless.

It is now possible to go on and give a revised axiomatic specification of the ADT Table, so that it no longer suffers from the horror of collapsing types. First, the operations "OK" and "IFE" must be expanded to handle sorts in general. This is done by subscripting the error value, "E", and the necessary operations, with the sort to which they belong. For example, the error value in the sort boolean will be denoted by $E_{boolean}$. It now becomes necessary to provide, for each sort "s", an error value "E_s" and a predicate "OK_s". This is so that each specification may assume the existence of error values for each type upon which it depends, without the danger that an undefined predicate or constant will be used.

Now assuming that OK_s and E_s have been added to the specification for each sort s, then the operator "IFE_s" can be defined as:

$$IFE_s(\text{true}, S_1, S_2) = S_1$$
$$IFE_s(\text{false}, S_1, S_2) = S_2$$
$$IFE_s(E_{boolean}, S_1, S_2) = E_s$$

where "S_1" and "S_2" are expressions of sort s. It is also necessary to assume the existence of derived operators which can be defined by:

$$OK_w(y_1, \ldots, y_n) = \wedge_{i=1}^{n} OK_{s_i}(y_i)$$

and

$$IFOK_{w,\,s}(y_1, \ldots, y_n, x) = IFE_s(OK_w(y_1, \ldots, y_n), x, E_s)$$

for $w = s_1 \ldots s_n$, where y_i is a variable of sort s_i and x is a variable of sort s.

ADT boolean

sorts boolean/boolean

comments

The values “true” and “false” are boolean constants.
The operator “not” provides the logical negative.
The operator “and” finds the logical conjunction.
The operator “or” finds the logical disjunction.
The operator “equal” returns a boolean value indicating the equality of the two boolean values.
The operators “IFE”, “OK” and “IFOK” behave as discussed in the text.

syntax

OK–ops

true:		$\rightarrow$	boolean
false:		$\rightarrow$	boolean
not:	boolean	$\rightarrow$	boolean
and:	boolean $\times$ boolean	$\rightarrow$	boolean
or:	boolean $\times$ boolean	$\rightarrow$	boolean
equal:	boolean $\times$ boolean	$\rightarrow$	boolean
IFE:	boolean $\times$ boolean $\times$ boolean	$\rightarrow$	boolean
OK:	boolean	$\rightarrow$	boolean
IFOK:	boolean $\times$ boolean	$\rightarrow$	boolean

error–ops

E		$\rightarrow$	boolean

semantics

declare b_1, b_2, b_3: boolean

axioms

OK–specs

(Ax-1) not(true) = false
(Ax-2) not(false) = true
(Ax-3) IFOK$_1$(b_1, not(not(b_1))) = IFOK$_1$(b_1, b_1)
(Ax-4) IFOK$_1$(b_1, and(true, b_1)) = IFOK$_1$(b_1, b_1)
(Ax-5) IFOK$_1$(b_1, and(false, b_1)) = IFOK$_1$(b_1, false)
(Ax-6) IFOK$_2$(b_1, b_2, and(b_1, b_2)) = IFOK$_2$(b_1, b_2, and(b_2, b_1))
(Ax-7) IFOK$_1$(b_1, or(true, b_1)) = IFOK$_1$(b_1, true)
(Ax-8) IFOK$_1$(b_1, or(false, b_1)) = IFOK$_1$(b_1, b_1)
(Ax-9) IFOK$_2$(b_1, b_2, or(b_1, b_2)) = IFOK$_2$(b_1, b_2, or(b_2, b_1))
(Ax-10) IFOK$_2$(b_1, b_2, equal(b_1, b_2)) =
IFOK$_2$(b_1, b_2, equal(b_2, b_1))

Figure 2.3. The type “boolean”.

(Ax-11) equal(true, true) = true
(Ax-12) equal(true, false) = false
(Ax-13) equal(false, true) = false
(Ax-14) equal(false, false) = true
(Ax-15) IFOK$_2$(b_1, b_2, IFE(true, b_1, b_2)) = IFOK$_2$(b_1, b_2, b_1)
(Ax-16) IFOK$_2$(b_1, b_2, IFE(false, b_1, b_2)) = IFOK$_2$(b_1, b_2, b_2)
(Ax-17) IFOK$_2$(b_1, b_2, IFE(E, b_1, b_2)) = IFOK$_2$(b_1, b_2, E)

error–specs
(Error-1) not(E) = E
((Error-2) and(E, b_1) = E
((Error-3) and(b_1, E) = E
((Error-4) or(E, b_1) = E
((Error-5) or(b_1, E) = E
((Error-6) equal(E, b_1) = E
((Error-7) equal(b_1, E) = E

OK–definition
(OK-1) OK$_1$(true) = true
(OK-2) OK$_1$(false) = true
(OK-3) OK$_1$(E) = false
(OK-4) OK$_1$(not(b_1)) = OK$_1$(b_1)
(OK-5) OK$_1$(and(b_1, b_2)) = and(OK$_1$(b_1), OK$_1$(b_2))
(OK-6) OK$_1$(or(b_1, b_2)) = and(OK$_1$(b_1), OK$_1$(b_2))
(OK-7) OK$_1$(equal(b_1, b_2)) = and(OK$_1$(b_1), OK$_1$(b_2))
(OK-8) OK$_1$(IFE(b_1, b_2, b_3)) =
and(and(OK$_1$(b_1), OK$_1$(b_2)), OK$_1$(b_3))
(OK-9) OK$_1$(OK$_1$(b_1)) = OK$_1$(b_1)

Figure 2.3. Continued.

If "e" is an equation involving distinct variables $y_1, \ldots, y_n$, and if y_i is of sort s_i, then the *arity* of e is:

$$w = s_1 \ldots s_n$$

As an application of the above approach, consider an axiom such as:

$$\begin{aligned} &\mathrm{IFOK}_{table\ index\ information,\ boolean}(\mathrm{tab, index\_1, info\_1,} \\ &\qquad\qquad \mathrm{empty(insert(tab, index\_1, info\_1)))} \\ &\quad = \mathrm{IFOK}_{table\ index\ information,\ boolean}(\mathrm{tab, index\_1, info\_1, false}) \qquad (*) \end{aligned}$$

which indicates that the variable "tab" is of sort table, "index_1" is of sort index and "info_1" is of sort information, and the expression:

$$\mathrm{empty(insert(tab, index\_1, info\_1))}$$

returns a result of sort boolean. If the predicates $\mathrm{OK}_{table}(\mathrm{tab})$, $\mathrm{OK}_{index}(\mathrm{index\_1})$ and $\mathrm{OK}_{information}(\mathrm{info\_1})$ all return true, then the above axiom (∗) is equivalent to:

$$\mathrm{empty(insert(tab, index\_1, info\_1))} = \mathrm{false}$$

where:

$$\begin{aligned} &\mathrm{tab} \neq \mathrm{E}_{table} \\ &\mathrm{index\_1} \neq \mathrm{E}_{index} \\ &\mathrm{info\_1} \neq \mathrm{E}_{information} \end{aligned}$$

If any of $\mathrm{OK}_{table}(\mathrm{tab})$, $\mathrm{OK}_{index}(\mathrm{index\_1})$ or $\mathrm{OK}_{information}(\mathrm{info\_1})$ does not hold, then the axiom (∗) reduces to the trivial statement:

$$\mathrm{E}_{boolean} = \mathrm{E}_{boolean}$$

Since every sort that is defined will need to use the IFE_s, OK_s and IFOK_s operators, and they will have exactly the same form in each case, their definition will be assumed to be that defined above and they will be dropped from the signature of each sort.

Hence, the description of Table can be given as shown in Figure 2.4. Similar definitions must also exist for boolean, index and information, because the specification in Figure 2.4 relies on their existence. The description in Figure 2.4 is identical to the specification given for Table earlier, except that the problem of collapsing types no longer exists.

Now that a set of axioms describing the operations on the ADT has been defined, it remains to be shown that the axioms satisfy the properties of consistency and sufficient-completeness (discussed informally at the beginning of this section). Although it is possible to show that these properties are satisfied by the axioms above, it may be impractical to prove that they are satisfied in the specification of a more complex ADT.

2.4 Verifying the Specification

2.4.1 Consistency

Having produced a set of axioms describing an ADT, the task of properly specifying the ADT is far from complete. In fact, the most difficult part is yet to come: proving consistency of the axioms, as introduced in Section 2.2. It must be shown that the axioms presented describe the ADT in a fashion that will result in only one correct interpretation for each expression of type Table. It can be shown that constructing such a proof is an undecidable problem for an arbitrary set of equations [51, 122].

To demonstrate inconsistency, it is sufficient to derive one contradictory statement. For example, if the axioms could be applied to a statement in a particular order until no further reduction is possible, to provide "a", say, by applying the axioms in one order,

ADT table [index,
information]
sorts table/ boolean, index, information

where index **has** equal: index × index → boolean

comments
The operator "new_table" creates an object of type Table.
The function "empty" tests to see if a Table object is equal to "new_table".
The operator "member" indicates whether or not a particular item is an index into the given Table object; i.e., it tests to see if there is an item in the Table object with the given index value.
The operator "insert" will insert an item into a Table object, replacing any existing item which has that index value.
The operator "remove" deletes the item with the specified index value if it is present. Otherwise, an error occurs.
The operator "search" returns the information associated with an index value if that index value is present. Otherwise, an error occurs.
The operator "alter" changes the information associated with an item in the Table if the relevant index value is present. Otherwise, an error is issued.

syntax
OK–ops

new_table:		→	table
empty:	table	→	boolean
member:	table × index	→	boolean
insert:	table × index × information	→	table
remove:	table × index	→	table
alter:	table × index × information	→	table
search:	table × index	→	information

error–ops

E_{table}:		→	table

semantics
declare tab: table
index_1, index_2: index
info_1, info_2: information
axioms
OK–specs
(Ax-1) empty(new_table) = true
(Ax-2) $\text{IFOK}_{table\ index\ information,\ boolean}$(tab, index_1, info_1,
empty(insert(tab, index_1, info_1)))
= $\text{IFOK}_{table\ index\ information,\ boolean}$(tab, index_1, info_1, false)

Figure 2.4. Revised abstract data type specification for "Table".

```
(Ax-3)  IFOK$_{index,\ boolean}$(index_1, member(new_table, index_1))
            = IFOK$_{index,\ boolean}$(index_1, false)
(Ax-4)  IFOK$_{table\ index\ index\ information,\ boolean}$(tab, index_1, index_2, info_1,
              member(insert(tab, index_1, info_1), index_2))
            = IFOK$_{table\ index\ index\ information,\ boolean}$(tab, index_1, index_2,
                                      info_1,
                 if equal(index_1, index_2)
                 then
                    true
                 else
                    member(tab, index_2)
                 end if)
(Ax-5)  IFOK$_{table\ index\ index\ information,\ table}$(tab, index_1,
                                      index_2, info_1,
              remove(insert(tab, index_1, info_1), index_2))
            = IFOK$_{table\ index\ index\ information,\ table}$(tab, index_1, index_2,
                                      info_1,
                 if equal(index_1, index_2)
                 then
                    tab
                 else
                    insert(remove(tab, index_2), index_1, info_1)
                 end if)
(Ax-6)  IFOK$_{table\ index\ index\ information,\ information}$(tab, index_1, index_2,
              info_1, search(insert(tab, index_1, info_1), index_2))
            = IFOK$_{table\ index\ index\ information,\ information}$(tab, index_1,
                                      index_2, info_1,
                 if equal(index_1, index_2)
                 then
                    info_1
                 else
                    search(tab, index_2)
                 end if)
(Ax-7)  IFOK$_{table\ index\ information,\ table}$(tab, index_1, info_1,
                 alter(tab, index_1, info_1))
            = IFOK$_{table\ index\ information,\ table}$(tab, index_1, info_1,
                 insert(remove(tab, index_1), index_1, info_1))
```

Figure 2.4. Continued.

(Ax-8) $\mathrm{IFOK}_{table\ index\ index\ information\ information,\ table}$(tab, index_1, index_2,
info_1, info_2,
insert(insert(tab, index_1, info_1), index_2, info_2))
= $\mathrm{IFOK}_{table\ index\ index\ information\ information,\ table}$(tab, index_1,
index_2, info_1, info_2,
if member(insert(tab, index_1, info_1), index_2)
then
alter(insert(tab, index_1, info_1), index_2, info_2)
else
insert(insert(tab, index_2, info_2), index_1, info_1)
end if)

error–specs

(Error-1)	alter(new_table, index_1, info_1) = E_{table}
(Error-2)	remove(new_table, index_1) = E_{table}
(Error-3)	search(new_table, index_1) = $\mathrm{E}_{information}$
(Error-4)	empty(E_{table}) = $\mathrm{E}_{boolean}$
(Error-5)	member(E_{table}, index_1) = $\mathrm{E}_{boolean}$
(Error-6)	member(tab, E_{index}) = $\mathrm{E}_{boolean}$
(Error-7)	insert(E_{tab}, index_1, info_1) = E_{table}
(Error-8)	insert(tab, E_{index}, info_1) = E_{table}
(Error-9)	insert(tab, index_1, $\mathrm{E}_{information}$) = E_{table}
(Error-10)	remove(E_{table}, index_1) = E_{table}
(Error-11)	remove(tab, E_{index}) = E_{table}
(Error-12)	alter(E_{table}, index_1, info_1) = E_{table}
(Error-13)	alter(tab, E_{index}, info_1) = E_{table}
(Error-14)	alter(tab, index_1, $\mathrm{E}_{information}$) = E_{table}
(Error-15)	search(E_{table}, index_1) = $\mathrm{E}_{information}$
(Error-16)	search(tab, E_{index}) = $\mathrm{E}_{information}$

OK–definition

(OK-1) OK_{table}(empty(tab)) = OK_{table}(tab)

(OK-2) $\mathrm{OK}_{table\ index}$(member(tab, index_1))
= and(OK_{table}(tab), OK_{index}(index_1))

(OK-3) $\mathrm{OK}_{table\ index\ information}$(insert(tab, index_1, info_1))
= and(and(OK_{table}(tab), OK_{index}(index_1)),
$\mathrm{OK}_{information}$(info_1))

(OK-4) $\mathrm{OK}_{table\ index}$(remove(tab, index_1))
= and(OK_{table}(tab), OK_{index}(index_1))

Figure 2.4. Continued.

$$
\begin{array}{ll}
\text{(OK-5)} & \text{OK}_{table\ index\ information}(\text{alter(tab, index\_1, info\_1)}) \\
& \quad = \text{and(and(OK}_{table}\text{(tab), OK}_{index}\text{(index\_1)),} \\
& \qquad \text{OK}_{information}\text{(info\_1))} \\
\text{(OK-6)} & \text{OK}_{table\ index}\text{(search(tab, index\_1))} \\
& \quad = \text{and(OK}_{table}\text{(tab), OK}_{index}\text{(index\_1))} \\
\text{(OK-7)} & \text{OK}_{table}\text{(new\_table) = true} \\
\text{(OK-8)} & \text{OK}_{table}(\text{E}_{table}) = \text{false}
\end{array}
$$

Figure 2.4. Continued.

and "b" by applying the axioms in another order, then the axioms are inconsistent if "a" is different from "b".

Proving consistency, on the other hand, can be done by several methods. The most common and widely used method is to construct a model of the ADT. If such a model can be written and formally verified to be correct, then the axioms are consistent. The drawback with this method is that it is possible to expend much time and effort in trying to construct a model that may not exist because the axioms are inconsistent. Even if such a model is produced, it must still be formally verified and shown to be consistent with the ADT in order to demonstrate consistency. This verification process is a laborious and error-prone task which is likely to take a substantial amount of time.

Another method for proving consistency is showing that the axioms, when treated as left-to-right rewrite rules, satisfy the Church-Rosser property [51, 122]. Informally, a set of rewrite rules satisfies the Church-Rosser property if, whenever a rule is applied to reduce a term and the resulting term is again reduced until no further reduction can take place, the final result is independent of the order in which the rules were applied. However, any such proof would involve considering each of the axioms in Figure 2.4 and demonstrating that the Church-Rosser property was satisfied. This method of proof is rigorous, but very detailed and difficult to produce, and will not be used here.

2.4.2 Sufficient-Completeness

By sufficient-completeness, as introduced in Section 2.2, it is meant that no expressions consisting of constants only (i.e., no variables) which operate on an object of the type of interest, such as Table, and which return an object of some other type, are without value or meaning. Proving that an algebraic specification is sufficiently-complete is also an undecidable problem [43, 51].

To define sufficient-completeness, we introduce two sets S and O such that S contains all the operations whose range is the type being specified and O contains those operations that map this type onto other types. If we now define a ground term to be a term that contains no variables, then we can state a definition for sufficient-completeness. For any abstract type T, and any axiom set A, A is a sufficiently-complete axiomatization of T, if and only if for every ground term of the form $o(e_1, ..., e_n)$, where o belongs to O, and each of the e_i is a ground term argument for the operator o, then there exists a theorem derivable from A of the form $o(e_1, ..., e_n) = u$ where u contains no operations on the type T.

By considering Figure 2.4 again, it can be seen that:

$$\begin{aligned} S = &\ \{ \text{ alter, remove, new\_table, insert } \} \\ O = &\ \{ \text{ empty, member, search } \} \end{aligned}$$

Sufficient-completeness for the axioms presented above will now be demonstrated. In this demonstration, let the notation:

$$\{\text{empty(tab)} \mid \text{tab} = \text{new\_table}\}$$

denote the set of all ground terms corresponding to the operation "empty" applied to a table called "tab", such that "tab" is equivalent to an invocation of the "new_table" operation. This means that "tab" is either the empty table as generated by a call to "new_table", or it is an object obtained by applying operations to a table such that the result is an empty table; that is to say, it is a member of the same equivalence class as "new_table". For example:

remove(insert(new_table,index_1,info_1),index_1)

is equivalent to:

new_table

because of axiom Ax–5.

By considering all possibilities, it is possible to show sufficient-completeness for the ADT specification as follows:

$$\begin{array}{lr}
\{\ \text{empty(tab)} \mid \text{tab} = \text{new\_table}\ \} = \{\ \text{true}\ \} & \text{by Ax-1} \\
\{\ \text{empty(tab)} \mid \text{tab} = \text{E}_{table}\ \} = \{\ \text{E}_{boolean}\ \} & \text{by Error-4} \\
\{\ \text{empty(tab)} \mid \text{tab} \notin [\text{new\_table, E}_{table}]\ \} = \{\ \text{false}\ \} & \text{by Ax-2}
\end{array}$$

$$\Rightarrow \{\ \text{empty(tab)} \mid \text{ for all possible values of tab}\ \} = \{\ \text{true, false, E}_{boolean}\ \}$$

Thus, it has been shown that the ADT specification is sufficiently-complete with respect to one of the members of the set O, namely "empty". The operation "member" is considered next; it is demonstrated that the property of sufficient-completeness holds for this operation also:

{ member(tab,index) | tab = new_table } = { false } by Ax-3
{ member(tab,index) | tab = E_{table} } = { $E_{boolean}$ } by Error-5
{ member(tab,index) | index = E_{index} } = { $E_{boolean}$ } by Error-6
{ member(tab,index) | tab $\notin$ [new_table, E_{table}] and index $\neq$ E_{index} and index is in tab } = { true } by Ax-4
{ member(tab,index) | tab $\notin$ [new_table, E_{table}] and index $\neq$ E_{index} and index is not in tab } = { false } by Ax-3, Ax-4

$\Rightarrow$ { member(tab,index) | for all possible values of tab and index } = { true, false, $E_{boolean}$ }

It now only remains to demonstrate the property of sufficient-completeness is also satisfied by the operation "search" to have shown that the property of sufficient-completeness is satisfied by the entire ADT specification. Thus, we finally consider the operation "search":

{ search(tab,index) | tab = new_table } = { $E_{information}$ } by Error-3
{ search(tab,index) | tab = E_{table} } = { $E_{information}$ } by Error-15
{ search(tab,index) | index = E_{index} } = { $E_{information}$ } by Error-16
{ search(tab,index) | tab $\notin$ [new_table, E_{table}] and index $\neq$ E_{index} index is in tab } = { [x | x $\in$ information] } by Ax-6
{ search(tab,index) | tab $\notin$ [new_table, E_{table}] and index $\neq$ E_{index} and index is not in tab } = { $E_{information}$ } by Ax-6, Error-3

$\Rightarrow$ { search(tab,index) | for all possible values of tab and index } = { [x | x $\in$ information] $\bigcup$ $E_{information}$ }

Thus, for each operation $o \in O$, there exists an expression u such that:

$$o(e_1, \ldots, e_n) = u$$

and u contains no operations on type Table. Hence, the ADT specification is sufficiently-complete.

2.4.3 An Alternative Approach

As an alternative to the approaches described in Sections 2.4.1 and 2.4.2, a slightly more informal method may be adopted, as is presented by Freidel [33]. Basically, this approach takes the two sets S and O from Section 2.4.2 and examines them closely. Freidel defines the set S to be the set of *constructors* (i.e., those operations which result in an object of the type of interest), and set O to be the set of *transfer operators* (those operations that return values other than the type of interest). Transfer functions are also referred to as *selectors.*

The abstract data type semantics define *equivalence classes* on the abstract data type objects, as discussed in Section 2.2. Formally, proofs are generally carried out by induction over *canonical forms* of the abstract data type objects; these canonical forms represent a selection of one member from each equivalence class. To simplify the ADT verification, the canonical terms are usually selected so that they are composed of only a subset of the constructors; these constructors are known as the *basic constructors.*

The set of constructors, S, for the ADT specification in Figure 2.4 was given in Section 2.4.2. By considering each of the elements of this set in turn, the basic constructors can be deduced. Note that axiom Ax-7 indicates that an object that contains the operator "alter" is equivalent to an object constructed without the operator "alter". Axiom Error-1 indicates that the "alter" operation can be replaced by "E_{table}" when applied to a "new_table". This implies that "alter" is not a member of the set of basic constructors. By application of the axioms Ax-5 and Error-2, it is possible to delete the operator "remove" from the set of basic constructors. It is not possible, however, to remove the operators "new_table" and "insert" and hence they are the only basic constructors. The canonical form for any object is composed of one "new_table" operation and zero or more "insert" operations. It now remains to analyze the transfer functions O with respect to the canonical expressions and demonstrate that the ADT is consistent and sufficiently-complete.

Induction over the set of canonical terms involves two cases for each operator: the initial case and the general case. For the operator "empty", there is only one axiom, Ax-1, that describes the initial case, and only axiom Ax-2 describing the general case. Note that, with respect to canonical terms, axiom Ax-1 refers only to the object "new_table" and axiom Ax-2 refers to one or more applications of the operator "insert" applied to a "new_table". Since "empty" returns only one value for each Table object, the axioms in Figure 2.4 are consistent and sufficiently-complete with respect to the operator "empty".

Next, consider the operator "member". Axiom Ax-3 describes the initial case and axiom Ax-4 the general case. Since there is only one axiom to describe the initial case and one to describe the general case, "member" can only return one value for each object of type Table. Hence, the axioms are consistent and sufficiently-complete with respect to the operator "member".

Axioms Ax-6 and Error-3 show that the "search" operator is defined for all Table objects. Since only axiom Error-3 relates "search" to the initial case and only axiom Ax-6 relates it to the general case, "search" returns only one value for each Table object. Thus, the axioms are consistent and sufficiently-complete with respect to the "search" operator.

2.5 The Use of Algebraic Specification Techniques in this Thesis

This chapter has presented two mechanisms for the definition of ADT's within an algebraic framework. The first, illustrated in Figure 2.1, is the simpler to read but not as formal and precise as the second one illustrated by Figure 2.4; the increased precision in the second relates particularly to handling of error terms. Both approaches provide the necessary degree of formalism required by the model to be presented in Chapter 3. For the purposes of this thesis, however, a choice has to be made. The simpler ADT specification (Figure 2.1) is selected because it is the easier to read, hence making the model developed in Chapter 3 easier to read and understand. Furthermore, the notation used in the ADT description of Figure 2.1 is typical of that used in the literature [44, 48, 52] and is widely known and understood. In addition, the approach forms the basis for integrated environments, such as ASSPEGIQUE [10, 11, 21], which support the development of large algebraic specifications through the PLUSS language. Other algebraic specification tools exist, each offering a complete range of operators for building specifications. These approaches all typically help in the development of an abstract data type specification, but none generates a complete implementation based on the description. These other specification techniques include CLEAR [18], OBJ3 [45, 74], Larch [54, 154] and OBSCURE [77, 78].

The information structure model for programming language semantics presented in the next chapter uses ADT's to provide a formal basis upon which to build. This differs from many other information structure models which formally define language semantics in terms of manipulations on data structures, but rely on an informal specification of these data structures. ADT's, however, suffer from certain limitations which restrict the model to the description of sequential programming languages; in particular, languages which involve concurrency need to employ a technique such as shared data abstractions (SDA's), which are discussed in Chapter 5.

Chapter 3

Describing Sequential Languages

3.1 Introduction

By employing ADT's as the basis for an information structure model, a formal underpinning for an operational semantic model is provided. This allows the development of a model of programming language semantics which covers both the static and dynamic semantic aspects of the language which is both formal and accurate. Structuring the model into layers allows varying amounts of information to be supplied, so that a progression from an abstract definition of the programming language semantics to a detailed description of the manipulations of the information structures is achieved. By supplementing each layer with a natural language description describing those aspects which are defined in deeper layers, it is then possible to provide a model that is useful to several classes of reader. Compiler writers and language designers have the detailed semantic information which they require by examining each layer of the model.

On the other hand, programmers will be able to examine the most abstract, topmost layer of the model, which describes the precise behaviour of each aspect of the language at a level of detail appropriate to the needs of programmers. Since each layer is accompanied by a natural language description of aspects defined more precisely in layers deeper in the model, a programmer is able to combine the formal definition with the informal description in order to answer simple queries regarding language semantics. Although the natural language narrative accompanying a layer is informal, and does not constitute part of the language definition, its presence is essential to service the need of one group of user (the programmer), while the formal definition is still useful to the compiler writer and the language designer. Ambiguities, as always, may arise in any informal natural language description, but it is hoped that by having the narrative do little more that verbalize the action of a formal aspect of the model, the language designer is able to avoid many potential problems.

Any reader of a language definition must then either follow each of the layers to the core of the model and hence gain a deep understanding of the programming language based on an operational semantic model, or the reader must depart from the formal definition and be content with the informal description of the formal definition. Such a transition from the formal to the informal model is a conscious decision which must be made by each user of the language definition. This transition will be desirable at different points for different groups of user, and even for different people within these groups. In essence, each reader may tailor the language definition to suit their needs.

A model developed in this way avoids many of the problems prevalent in current language definition techniques which subsequently restrict their use to only one class of user, as discussed in Chapter 1. By avoiding these difficulties, the model is able to provide for the needs of diverse groups of readers, with diverse interests, and provide them all with a single language definition; this avoids the provision of an official language definition coexisting with supposedly equivalent descriptions targeted at different groups.

Layer 3	Definition of Language Features	Outermost Layer
Layer 2	High Level Operations	Middle Layer
Layer 1	Algebraic Specifications of ADT's	Innermost Layer

Figure 3.1. Layering an operational semantic model.

3.2 The Structure of the Multi Layered Model

The model described in this chapter has three layers which describe varying levels of detail relating to the model; these layers are illustrated in Figure 3.1. The core, or innermost layer, of the model is the layer which describes all the ADT's used within the model and provides a precise description of the information structures. This layer contains all the ADT specifications and ideally should be accompanied by proofs of consistency and sufficient-completeness for each. This layer, although large and seemingly complex, is almost completely reusable. If a different language description is required, it is possible to reuse many of the ADT definitions used in earlier models of other languages. As a result, there is little wastage of effort in this layer. The notation used for the description of the ADT's is that developed in the previous chapter.

The second layer, which builds on top of the first layer, contains a number of shorthand notations for common ADT manipulations; these are known as *high level operations.* The introduction of this layer eliminates the need to repeat large sections of common ADT manipulations and hence enhances readability.

The outermost layer of the model is the *user interface.* This layer employs the layers beneath it to produce a description of how the semantics of the programming language features currently being defined affect the information structures which are built up by the layers beneath it, and it presents the most abstract view of the static and dynamic semantics of the programming language to the reader. It is this layer, whose accompanying natural language description of the lower layers, provides a panoramic view of the programming language semantics. This layer will be of most use to programmers wishing to gain an understanding of the language and most programmers will need to go no further.

The natural language description which accompanies each layer does not constitute a part of the language definition, but is simply a supplement to it. The formal language definition consists of all of the layers collectively. The natural language description is based on each of the layers within the model and if any ambiguities occur within the informal description, or between the informal description and the formal description, then the ambiguity can be resolved by examining the formal description. The natural language narrative merely provides documentation which is easier for programmers to digest; this is necessary to make the definition useful to all potential users.

Layer 1 of the model, the ADT specifications, has a firm mathematical background and provides the solid foundation upon which the formal semantic model can be developed. The other layers of the model maintain the formal nature of the model by using constructs which themselves have formal definitions. The three constructs of prime importance in Layers 2 and 3 of the model are:

- sequencing,
- selection, and
- iteration.

The model uses the sequencing operator ";" to define an ordering on the execution of actions. The sequence:

$$\ll S_1 \gg ; \ll S_2 \gg ;$$

simply means to evaluate action $\ll S_1 \gg$ first and, on its completion, evaluate the action $\ll S_2 \gg$. The ";" operator is considered to be a basic operator within the model and its formal definition can be found in [46, 112]; it is used as an action terminator throughout this thesis, to indicate the end of each action.

Selection is achieved through the **if-then** and **if-then-else** constructs. The semantics of such constructs has been defined axiomatically in [49] and has the conventional meaning; hence:

$$\textbf{if} \ll E \gg \textbf{then} \ll S \gg \textbf{end if} ;$$

involves the evaluation of the logical expression $\ll E \gg$ whose truth value is then used to determine whether the action sequence $\ll S \gg$ is to be executed. (Note that the actions in the action sequence $\ll S \gg$ will be separated and terminated by ";".) A version of the construct exists which allows a choice to be made between two action sequences. This more general form is given as:

$$\textbf{if} \ll E \gg \textbf{then} \ll S_1 \gg \textbf{else} \ll S_2 \gg \textbf{end if} ;$$

In this form, the logical expression $\ll E \gg$ is evaluated and if found to be true, the action sequence $\ll S_1 \gg$ is executed and the action sequence $\ll S_2 \gg$ is ignored. If the expression $\ll E \gg$ evaluates to false, then the reverse occurs: the action sequence $\ll S_1 \gg$ is ignored and the action sequence $\ll S_2 \gg$ is evaluated. The **if-then** construct is clearly just a special case of the **if-then-else** construct.

Repetitive constructs can be introduced into the model and defined in terms of the **if-then-else** construct, hence retaining the formal nature of the model. Two forms

```
while ≪E≫
loop
    ≪S≫
end loop ;

          ≡ if ≪E≫ then
                ≪S≫
                while ≪E≫ loop
                    ≪S≫
                end loop ;
            end if ;
```

Figure 3.2. Repetition (while-loop) defined in terms of selection.

```
repeat
    ≪S≫
until ≪E≫ ;

      ≡ ≪S≫
        while ≪E≫ loop ≪S≫ end loop ;
```

Figure 3.3. Repetition (repeat-until) defined in terms of selection.

of indefinite repetition are introduced and defined in Figures 3.2 and 3.3. Figure 3.2 introduces the **while** loop, such that the boolean expression ≪E≫ is evaluated and, if found to be true, the action sequence ≪S≫ is evaluated before re-evaluating the expression. If the expression evaluates to false, then the action sequence following the **while** loop is evaluated. The **repeat** loop introduced in Figure 3.3 works in a similar fashion, but the condition is only evaluated after the action sequence has been evaluated.

A final form of repetition is presented in the form of a **for** loop. Such a construct evaluates for some fixed number of iterations and its form is shown in Figure 3.4. The definition in the figure indicates that ≪variable≫ commences with the value determined by ≪lower bound≫ and steps through each value in the range of specified values in a monotonically increasing manner with a step size of one until the value

```
for $\ll$variable$\gg$ in $\ll$lower bound$\gg$ .. $\ll$upper bound$\gg$
loop
    $\ll$S$\gg$
end loop ;

      $\equiv$ $\ll$variable$\gg$ := $\ll$lower bound$\gg$;
        while $\ll$variable$\gg$ $\leq$ $\ll$upper bound$\gg$ loop
          $\ll$S$\gg$
          $\ll$variable$\gg$ := $\ll$variable$\gg$ + 1;
        end loop ;
```

Figure 3.4. The form and definition of the for-loop.

associated with ≪variable≫ exceeds ≪upper bound≫. If ≪upper bound≫ is initially less than ≪lower bound≫, then the entire **for** construct has no effect. Throughout the model, ≪lower bound≫ and ≪upper bound≫ are integer values.

Figure 3.4 introduces the notion of *variable*, which is simply a name to stand for some value, and the notion of *assignment* (":="). These concepts are well-known and their semantics here is as expected in any conventional programming language. Hence, the statement:

≪variable≫ := ≪variable≫ + 1;

takes the value associated with ≪variable≫ and increments it by one before associating the new value with ≪variable≫. Formal definitions of this concept are well covered in the literature using various techniques (e.g., see [112]).

Since each of these constructs has a formal mathematical basis, their use in an operational semantic model preserves the formalism achieved through the use of abstract data types. The net result is an information structure model of the semantics of a programming language such that the model is formal in nature.

3.3 A Model of Data Control

As an example of an information structure model using the layered approach, a model of the data control aspect of the programming language Pascal is presented. Data control, following Pratt [116], is concerned with language features which govern access to the data objects (such as variables) of a program. Pascal has several such language features and these will be described in this section. These are the declaration of local variables, scope rules, value parameters, variable parameters, functional and procedural parameters, and the return of values from functions; for example, the problem considered in Section 1.2.1.2 stems from a question relating to data control in Pascal. The information structure model to be presented in this section will precisely describe each of the data control aspects of Pascal.

Very few semantic models have concentrated on, or even provided adequate descriptions of, data control in programming languages. Exceptions include Johnston's Contour Model [67], Smith's Accessing Graph Model [131], Reiss' ACORN project [118], Molinari and Johnson's enhancment of Reiss' work [96] and Marlin's model of data control [82, 83], later refined in [85]. The model developed in this chapter is based on the informal model introduced in [82, 83]; the necessary formalism is introduced into the model by the application of the techniques employed in [85], introducing abstract data types to formally define the information structures manipulated.

As already seen, there are difficulties in producing compilers to implement the demands of the language definition. The model developed here will not attempt to alter the semantics of Pascal, but instead will attempt to match the description of the Pascal definition [17] as best it can. Such an approach serves to highlight the difficulties introduced into Pascal as a result of using natural language to define the programming language semantics.

This model of the data control aspect of Pascal is an information structure model [150], in which the manipulation of information structures is used to describe the relevant features. The information structures are built up of abstract data types and as such have a formal basis that can be shown to be consistent and sufficiently-complete (recall Chapter 2). The use of abstract data types is also the key to the layering of the description depicted earlier in Figure 3.1 and discussed in Section 3.2.

3.3.1 Embedding Semantics Within a Syntactic Description

As already discussed in Chapter 1, the syntax of a language is readily and accurately described in some form of BNF. As this notation is widely understood, the model described mixes semantic definitions with a syntactic definition expressed using BNF to provide a comprehensive language definition with regard to data control.

For example, it is possible to embed a description of the effect of scope rules into the BNF syntactic definition of Pascal presented in [17]; a fragment of such a mixed syntactic/semantic description is shown in Figure 3.5. In this figure, the invocation of a semantic routine is shown in bold face, delimited by "%%". In the model, a semantic routine performs some manipulations on the relevant information structures. In the case of the semantic routine involved in Figure 3.5, the intention is to inherit all the known entities from the nearest textually enclosing block that have not been redefined within the current block.

One difficulty with the approach used in Figure 3.5 is that it may not be possible to specify the semantics of language features using this approach if one-pass analysis of the source text is assumed. An illustration of this difficulty in the context of Pascal is

```
<program> = <program-heading> ";" <program-block>
<procedure-and-function-declaration-part> =
          {(<procedure-declaration> | <function-declaration>) ";"}
<procedure-declaration> =
          <procedure-heading> ";" <directive>
          | <procedure-identification> ";" <procedure-block>
          | <procedure-heading> ":" <procedure-block>
<function-declaration> = <function-heading> ";" <directive>
          | <function-identification> ";" <function-block>
          | <function-heading> ":" <function-block>
<program-block> = <block>
<procedure-block> = <block>
<function-block> = <block>
<block> = <label-declaration-part>
          <constant-definition-part>
          <type-definition-part>
          <variable-declaration-part>
          <procedure-and-function-declaration-part>
          %% scope-rules %%
          <statement-part>
```

Figure 3.5. Enhancing syntax with semantics.

given in Figure 3.6. The reference to "P2" at line 6 (in the statement marked "{#}") should be regarded as invalid because the meaning of "P2" at the level of the block for procedure "P" is supplied by the declaration at line 8; furthermore, the reference to procedure "P2" at line 6 is invalid because the declaration occurs after the first use. The correct semantics of Pascal's scope rules can only be captured a two-pass description. The modified syntactic/semantic description is given in Figure 3.7, which shows the changes which must be made to the description of "<block>" given earlier in Figure 3.5.

The two semantic routines mentioned in Figure 3.7 have different purposes. The semantic routine "Scope-Rules-Pass-1" is used to inherit all the known names at that point from the nearest textually enclosing block. It simply takes those names that were declared local to the parent block and inherits them as nonlocal entities, providing

```
program EXAMPLE;

     var P2: integer;

     procedure P;

          procedure P1;
               begin {P1}
                    P2 := 2 {#}
               end; {P1}

          procedure P2;
               begin {P2}
               end; {P2}

          begin {P}
          end; {P}

     begin {EXAMPLE}
     end. {EXAMPLE}
```

Figure 3.6. A Pascal example.

```
<block> = %% Pass 2: Scope-Rules-Pass-2 %%
          <label-declaration-part>
          <constant-definition-part>
          <type-definition-part>
          <variable-declaration-part>
          <procedure-and-function-declaration-part>
          %% Pass 1: Scope-Rules-Pass-1 %%
          <statement-part>
```

Figure 3.7. Two-pass semantic description.

that the name has not been redefined within the current block. Hence, in the first pass, names are only inherited from the nearest textually enclosing block. In the example in Figure 3.6, this would mean that at line 5, the name "P2" would not be inherited as it is not yet declared in the nearest textually enclosing block ("P") and names are not inherited from any other block in the first pass. However, by the time that line 11 is reached in the first pass, procedure "P2" will be known to be local to procedure "P". The second pass makes use of information structures built during the first pass and propagates nonlocal declarations into inner blocks. Consequently, it is the responsibility of the semantic routine "Scope-Rules-Pass-2" to inherit into each block all accessible names that were not declared local to the nearest textually enclosing block.

There are many ways in which semantic issues such as scope rules can be handled correctly; using a multi-pass technique, such as that described above, is just one. Its use here highlights the advantages of a multi-pass model, as it makes descriptions easier to write and describes the semantics in a clear fashion, so that all users and implementors of the language are aware of the complexities and the nature of that aspect of the language semantics.

3.3.2 Abstract Data Type Definitions – the Formal Foundation

The first (innermost) layer in the model consists of abstract data type (ADT) specifications. This layer provides the necessary precise foundations for the model. The ADT's are specified using an algebraic technique similar to that adopted by Guttag *et al.* [51, 55], but making use of the initial algebra approach advocated by the ADJ group [41, 44], as discussed in Chapter 2. An example of an ADT specification typical of the model is given in Figure 3.8; this specification defines a list data type. This list specification is parameterized with respect to the sort of data stored within it, as indicated by the parameter "item". The **sorts** clause in the specification lists all the types used within the axiomatic definition. This example defines the data type "List" and makes use of the sorts "item" and "boolean". The syntax of the operations defined for the ADT are specified next: the name of the operation is given followed by the sorts of objects on which it operates, finally specifying the result type of the operation.

The semantics of the operations are defined in the section labelled **semantics** of Figure 3.8. This section, as previously discussed in Section 2.2, begins by declaring variables to stand for arbitrary objects of the types specified. No assignments are made to these variables – they are simply names which represent objects of the various types. Finally, the axioms are given; these provide the meaning associated with the operations of the ADT. Proofs of consistency and sufficient-completeness have been constructed for the ADT specifications used in the model of data control; these specifications can thus be regarded as a solid foundation on which to build the remainder of the model. A further example is given in Figure 3.9; this describes a type, used latter in the model, which describes links to table entries.

3.3.3 The Information Structures and Their Operations

As mentioned earlier, the model presented is an information structure model. In such models, the semantics of language features are specified in terms of transformations on information structures representing (aspects of) the state of a program in the language. The transformations describing the semantics of data control features occur within the

ADT List [item]
sorts List/item, boolean

comments

The operator "new_list" creates an object of sort List.
The operator "empty_list?" tests to see if a List object is equal to "new_list".
The operator "add_to_list" will insert an item into a List object at the tail of the List object.
The operator "head_of_list" returns the item at the head of the List object.
The operator "tail_of_list" discards the item at head of List object and returns the resultant List object.

syntax

new_list:		$\rightarrow$	List
empty_list?:	List	$\rightarrow$	boolean
add_to_list:	List $\times$ object	$\rightarrow$	List
head_of_list:	List	$\rightarrow$	object $\bigcup$ $\{\mathit{error}\}$
tail_of_list:	List	$\rightarrow$	List $\bigcup$ $\{\mathit{error}\}$

semantics

declare list_1: List
obj_1, obj_2: object

axioms

(1) empty_list?(new_list) = true
(2) empty_list?(add_to_list(list_1, obj_1)) = false
(3) head_of_list(new_list) = *error*
(4) head_of_list(add_to_list(new_list, obj_1)) = obj_1
(5) head_of_list(add_to_list(add_to_list(list_1, obj_1), obj_2)) =
 head_of_list(add_to_list(list_1, obj_1))
(6) tail_of_list(new_list) = *error*
(7) tail_of_list(add_to_list(new_list, obj_1)) = new_list
(8) tail_of_list(add_to_list(add_to_list(list_1, obj_1), obj_2)) =
 add_to_list(tail_of_list(add_to_list(list_1, obj_1)), obj_2)

Figure 3.8. An abstract data type describing a FIFO list.

```
ADT Link_Type   [group,
                 item,
                 link_kind]
sorts Link_Type/group, item, link_kind
```

comments

This ADT describes a link between two objects. Links can only be followed in one direction and are not visible from the other direction.

In the case of the data control model, the link points to an item within a group of items, and so the name of the group and the item within that group must be specified. A link of type "link_kind" is then established, where "link_kind" indicates the access rights along that link; these access rights include RW (Read-Write), RO (Read-Only) and WO (Write-Only). The operations are:

- "new_link" creates a new Link_Type object.
- "link" sets up a link to point to the required item in a particular group of items.
- "follow_group" returns the group of items to which the link points.
- "follow_item" returns the item within this group to which the link actually points.
- "kind_of_link" indicates what kind of link it is.

syntax

```
new_link:                                          →  Link_Type
link:            Link_Type × group × item × link_kind
                                                   →  Link_Type
follow_group:    Link_Type                         →  group ⋃ {error}
follow_item:     Link_Type                         →  item ⋃ {error}
kind_of_link:    Link_Type                         →  link_kind ⋃ {error}
```

semantics

```
declare   link_1: Link_Type
          group_1: group
          item_1: item
          kind_1: link_kind
```

axioms

```
(1)  link(link_1, group_1, item_1, kind_1) =
           link(new_link, group_1, item_1, kind_1)
(2)  follow_group(new_link) = error
(3)  follow_group(link(link_1, group_1, item_1, kind_1)) = group_1
(4)  follow_item(new_link) = error
(5)  follow_item(link(link_1, group_1, item_1, kind_1)) = item_1
(6)  kind_of_link(new_link) = error
(7)  kind_of_link(link(link_1, group_1, item_1, kind_1)) = kind_1
```

Figure 3.9. The abstract data type Link_Type.

third and final layer of the model. The role of the second layer is to specify the nature of the information structure and to provide some high-level primitives in terms of which the transformations can be formulated; both the information structure and the high-level primitives are defined using the ADT's specified in the first layer.

Before discussing the information structure required for the description of data control in Pascal, there is another important data structure which must be introduced. This data structure, known as the ***static environment***, represents the static aspects of a Pascal program, recording the names of identifiers and associated attributes for each block; this data structure corresponds to what is normally called the ***symbol table*** in a compiler.

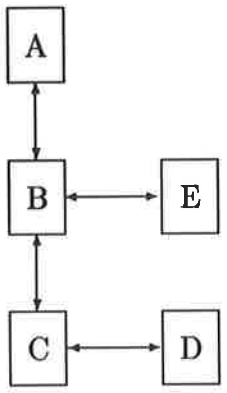
Even though Pascal does not allow the user to overload identifiers, the language itself does precisely this in the case of functions. For each function defined in a Pascal program, there is a corresponding function pseudo-variable with the same name. As a result of this inconsistency, it is necessary to represent the symbol table as a table, indexed by identifier name and where each element is a secondary table. This secondary table is indexed by the kind of the object (e.g., variable, function, function pseudo-variable, etc.) and stores the attributes associated with the object that has that name and that kind. This complicates the discussion of Pascal considerably, but highlights a conceptual difficulty encountered by someone learning and using the language Pascal.

The table representing the symbol table is itself a member of a structure called *symbol_table_info*; there is one such data structures for each block in the program and, apart from the symbol table, it also records the name of the block and other information. As explained in the previous section, it is necessary for the model presented here to operate in a multi-pass fashion in order to be able to correctly capture the semantics of data control in Pascal. Thus, it is necessary to build a symbol_table_info object as each block is encountered and store the results in some manner, so that the information is available in subsequent passes. Figure 3.10(a) shows a Pascal program containing a number of nested procedures; Figure 3.10(b) provides a pictorial representation of the corresponding static environment. The box labelled "A" at the top of the diagram in Figure 3.10(b) corresponds to the program "A" of Figure 3.10(a). Nested within the program "A" are the procedures "B" and "E"; this is illustrated in the diagram by having boxes labelled "B" and "E" at the next level. Procedure "B" occurs textually before procedure "E" in the code and as a result is callable from procedure "E", whilst procedure "B" is unaware of the presence of procedure "E"; this is reflected in the diagram by having box representing procedure "B" to the left of the box representing procedure "E". Similarly, procedures "C" and "D" are nested within procedure "B" and this is reflected diagrammatically by placing boxes representing these procedures at the next level of the diagram. Since procedures "C" and "D" are local to procedure "B", they are linked to procedure "B" in the diagram. Procedure "E" has no nested procedures, and as such there are no boxes extending from the box representing procedure "E" in Figure 3.10(b). Because of the nature of scoping in Pascal (which is oriented towards one-pass compilation), the tree representation used emphasizes the ordering of siblings at a particular block level. It is this tree structure which is known as the static environment, and each node is said to be of type *static_information.*

The information structure used in the description of the semantics of data control in Pascal represents the data control aspects of an executing Pascal program and is called the *dynamic environment.* It is much simpler than the data structure used in the static environment, as it consists simply of a list of instances called *activation*

```
program A;

  procedure B;

    procedure C;
      begin {C}
      end; {C}

    procedure D;
      begin {D}
      end; {D}

    begin {B}
    end; {B}

  procedure E;
    begin {E}
    end; {E}

  begin {A}
  end. {A}
```

(a)



(b)

Figure 3.10. A Pascal program and its static environment.

instances. Each instance records the information necessary to identify the block to which it corresponds and a symbol table object. When an instance of any block is created as a result of the semantics of an appropriate language feature, the contents of the symbol table object for the instance are initially an exact copy of the contents of the corresponding symbol table information object stored in the static environment for that block. As other language features are encountered, the contents of the symbol table in the instance may change.

In Pascal, some of the names accessible to a block instance do not correspond to objects introduced by that instance, but rather they stand for objects belonging to other instances. An example of this is the way in which formal variable parameters stand for the actual parameter variables with which they are associated; another example occurs when access to an object is inherited via the scope rules. Such names are described by ***non-defining*** objects in the model. On the other hand, names standing for objects introduced by the particular instance concerned are described in the model by ***defining*** objects. All non-defining objects are linked to some defining object throughout the former's lifetime. Some of these links are established in the static environment, as it is known then which objects they stand for; this is the case for objects that are inherited into a block via scope rules. For other non-defining objects, such as those for variable parameters, the links must be established in the dynamic environment. Various types of links are used, depending on the kind of access implied by the relationship between the non-defining object and the defining object. Two access rights to objects in a Pascal program can be distinguished: read (R) and write (W). The four subsets of these rights, namely:

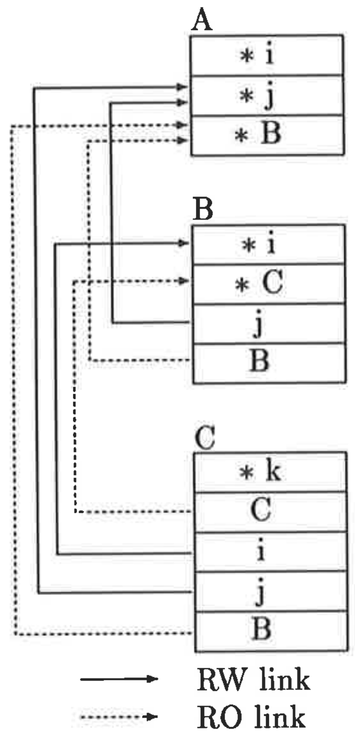
$$RW = \{R,W\} \qquad RO = \{R\} \qquad WO = \{W\} \qquad NA = \{\,\}$$

are all useful in describing the kind of access which applies to a particular object in a particular block, and in describing the kind of access permitted by a link. These links correspond to the ADT Link_Type given previously in Figure 3.9.

Figure 3.11(b) shows the dynamic environment during the execution of the program in Figure 3.11(a), at the point corresponding to the line marked "{#}" in Figure 3.11(a). Each of the instances contains a symbol table listing all entities known to each block (predefined names being omitted from the figure for clarity). Symbol table entries labelled with "*" represent defining objects. Note that links emanate from all non-defining entries and lead to a defining entry in each case. The links depicted in Figure 3.11(b) show that, in this case, RW access is propagated for variables and RO access is propagated for procedures.

```
program A;
  var
    i, j: integer;
  procedure B;
    var
      i: integer;
    procedure C;
      var
        k: integer;
      begin {C}
        {#}
      end; {C}
    begin {B}
      C;
    end; {B}
  begin {A}
  end. {A}
```

(a)



(b)

Figure 3.11. The dynamic environment for a Pascal program.

3.3.4 High Level Operations

As mentioned earlier, the second layer of the data control model includes some *high-level operations* (HLO's), which are used in the specification of the transformations in the third layer. The HLO's hide much of the detailed manipulation of the ADT's introduced in the first layer from the view of a reader of the third layer; the result is much shorter semantic descriptions of the data control aspect of Pascal than would otherwise have been attained, but without any loss of precision overall. By providing some natural language narrative with each HLO, the majority of the users of a specification will not need to examine the ADT specifications in detail. The formal model should constitute the definition of the programming language under consideration, and the natural language commentary should be regarded as an aid, nothing more.

```
member_of_symbol_table(a: static_information;
                       s: string) =
    member_of_table(current_block(return_info(a)), s);
```

Figure 3.12. The high-level operation **member_of_symbol_table.**

The routines presented in the second and third layers of the model make use of several control constructs, such as selection and repetitive constructs. Each of these constructs should be defined formally in order to retain the degree of formality obtained so far. The **if-then-else** construct can be defined axiomatically [49]; this then allows a formal definition of the **repeat-until, while-do** and **for** constructs in terms of the **if-then-else** construct, as shown in Section 3.2. In order to use a **for** loop construct effectively over arbitrary ADT's, it is necessary to introduce an operation to return the size of a data structure, and a mechanism whereby each element within the ADT can be accessed by its position within the data structure. This can be clearly seen in the examples to be presented in Section 3.3.5. The notation used for these constructs within the model draws on the notation and semantics of the corresponding constructs in the Ada programming language [141].

A data control model for a realistic language, such as Pascal, uses a great many higher level operations. As a result, only a limited number can be presented here. Section 3.3.5 uses the HLO's explained below in several places and consequently they have been selected as illustrations.

The first of the HLO's we will consider is **member_of_symbol_table**, defined in Figure 3.12. This HLO takes two parameters, the first ("a") is an object of type static_information, introduced in Section 3.3.3, and the second parameter ("s") is a string representing the identifier about which the enquiry is being made. This HLO returns a boolean result, reflecting whether the string corresponds to one of the entries in the symbol table. From Figure 3.12, it can be seen that several ADT operations are used in the definition of the HLO. The operation "return_info" is applied to the parameter "a" to yield an object of type symbol_table_info. The operation "current_block" is then applied to this object, giving a table object, namely the symbol table. Application of the operation "member_of_table", which takes a table and a string (the key in this case) and returns a boolean value, completes the HLO. Even from this simple example, it can be seen that the introduction of HLO's into the description can produce a more readable document than would otherwise be obtained.

```
add_new_info(t: static_information;
             s: string;
             k: kind;
             a: attribute_type) =
   if not(member_of_symbol_table(t, s))
   then
     define_info(t, define_current_block(return_info(t),
            insert_into_table(current_block(return_info(t)), s,
            insert_into_table(new_block, k, a))));
   else
     define_info(t, define_current_block(return_info(t),
            alter_table(current_block(return_info(t)), s,
            insert_into_table(associated_attributes(
                current_block(return_info(t)), s), k, a))));
   end if;
```

Figure 3.13. The high-level operation **add_new_info**.

Another HLO used in the model is **add_new_info**, given in Figure 3.13. This HLO is used to add new information into the symbol table. It takes four parameters: "t" of type static_information, "s" of type string (representing the identifier to be inserted), "k" representing the kind of object to be inserted, and "a" for the attributes associated with an object of this name and kind. Since Pascal overloads function names in the manner described earlier, the model is more complex than it would be for a language such as Modula-2 which uses a **return** statement. Consequently, the HLO must first ascertain if the identifier "s" is already present in the symbol table "t". If it is not, then it is inserted with relative ease. However, if it is already in the symbol table, then the secondary table associated with the identifier in the symbol table must be altered to take the additional information; this is the case for overloaded identifiers.

3.3.5 The Semantic Descriptions

The third and final layer of the data control model describes the semantics of the relevant features of the language concerned (Pascal, in this case). The data control aspect of Pascal covers local declarations, scope rules, value parameters, variable parameters, and procedure and function names passed as parameters.

3.3.5.1 Local Declarations

During the first pass of the description of Pascal, the analysis of declarations causes identifiers to be stored in a list until sufficient information is known about them to insert them into the symbol table associated with the current block. For example, by the time the semicolon is reached in the declaration "**var** i, j, k: integer;", it is known that the identifiers being declared are "i", "j" and "k", they are all variables (their kind) and are of type integer. It simply remains to insert the information into the symbol table. Inserting information into the symbol table means that we must know all of the attributes associated with that identifier, or at least as much as is possible at that point. In the case of forward declarations of procedures and functions, the information is recorded in the symbol table and the attribute "forward" is set. Later, when the declaration is completed, the attributes associated with the identifier can be updated. This approach also allows error handling facilities to examine each symbol table at the end of the declaration sections within each current block and ensure there are no outstanding forward declarations.

As the object is declared, its name, kind and type are easily obtained. All names introduced in a local declaration are of course "local" to the block in which they are declared. However, depending on the kind of object, a defining or a non-defining entry may be used. In fact, all objects except variable parameters, and functional and procedural parameters are declared as defining entities. These forms of parameters are exceptions because the name is not associated with a storage location or the point of definition of the relevant object; they are simply alternative names for objects declared elsewhere. Objects may also have different access rights defined for them, and this property is again related to the kind of the object concerned. Functions, procedures, as well as functional and procedural parameters, are defined as having RO access; variables, value parameters and variable parameters are defined as having RW access (as their values may be read as well as altered within the block in which they are defined).

The semantic routine shown in Figure 3.14 handles local declarations in Pascal. It is parameterized with respect to the name of the identifier being declared, its type, kind, definition, access rights and a flag to indicate if the object is being declared forward or not. The routine begins by checking if the identifier "s" is known to the symbol table for the current block. If not, then an attribute record is defined that records all the necessary information for an identifier: name, type, kind, access rights, whether it was declared local or nonlocal, whether it is a defining or non-defining entry, perhaps a link field indicating another entity to which it is linked, and a flag indicating if it is a forward declaration. The current block, "this_block", can then be updated. If the identifier was already present in the symbol table, then it may be a forward declaration

```
Local-Declaration(s: string;
                  t: type;
                  k: kind;
                  d: definition;
                  a: access_rights;
                  forward: boolean) =

 -- Check to see if this identifier is already known in this block.
 if not(member_of_symbol_table(this_block, s))
 then
   -- As it has not already been declared, it may be added
   -- to the symbol table for this block.
   attributes ← define_attributes(s, t, k, a, local, d, new_link, forward);
   -- update the static data structure.
   add_new_info(this_block, s, k, attributes);
 else
   -- Gain access to the symbol table object.
   tab_info ← get_info_via_ident_from_sym_tab(this_block, s);
   if member_of_info_tab(tab_info, k)
   then
     -- Handle completion of forward declarations.
     if and(declared_forward(get_attributes(tab_info, k)),
            not(forward))
     then
       attributes ← define_attributes(s, t, k, a, local, d, new_link, forward);
       -- Ensure attributes of forward declaration match current declaration.
       if match_attributes(attributes, get_attributes(tab_info, k))
       then
         -- update the static data structure
         alter_info(this_block, s, k, attributes);
       else
         Error("Attributes do not match.");
       end if;
     else
       Error("Error in forward declaration.");
     end if;
   else
     Error("Identifier previously declared to be of a different kind.");
   end if;
 end if;
```

Figure 3.14. Local declarations.

that is being completed. This is checked by retrieving the attributes associated with the identifier. Since overloading of identifiers is not generally allowed in Pascal, then the table of information associated with the identifier must contain only a single entry. If this entry represents an identifier which had been declared forward earlier, then the associated attributes are updated, otherwise an error message may be issued.

3.3.5.2 Scope Rules

Perhaps the most obvious data control aspect in a block structured language such as Pascal is that of scope rules. As explained earlier, a two-pass model is required to adequately describe the semantics of Pascal's particular form of implicit scope rules. Consequently, the definition of the scope rules of Pascal is divided into the two stages depicted in Figures 3.15 and 3.16. The first pass, depicted in Figure 3.15, inherits all entities that were declared local to the parent block and not redefined in the current block and the second pass, shown in Figure 3.16, inherits all the nonlocal entities that have been inherited into the parent block, again provided that they were not redefined in the current block.

The transformation **Scope-Rules-Pass-1**, given in Figure 3.15, commences by locating the parent block in the static environment. If it has a non-empty symbol table, then each identifier in this symbol table is considered in turn. If the identifier currently being considered is not a member of the symbol table for the current block, then it may be inherited into the current block's symbol table after it has been modified to represent a nonlocal, non-defining entity as far as the current block is concerned. The link field is also modified so that it points to the object that is being inherited from the parent block.

```
Scope-Rules-Pass-1 =

  -- Find the nearest textually enclosing block; this is called the parent block.
  locate_parent;
  if not(empty_symbol_table?(parent))
  then
    -- For every object in the symbol table.
    for i in 1 .. size_of_symbol_table(parent)
    loop
      -- Iterate through the symbol table of the parent block.
      -- Get the identifier.
      identifier ← get_ident_from_sym_tab(parent, i);
      if not(member_of_symbol_table(this_block, identifier))
      then
        -- If the identifier is unknown to this_block, then it may be inherited.
        tab_info ← get_info_from_sym_tab(parent, i);
        -- Must cycle through secondary attribute table associated with
        -- each identifier of the symbol table.
        for j in 1 .. size_of_info_table(tab_info)
        loop
          -- Set up the attributes, etc., properly.
          attributes ← get_attributes_from_info_table(tab_info, j);
          kind ← get_kind_from_info_table(tab_info, j);
          link ← link_to(parent, identifier, kind, access_rights(attributes));
          -- Specify that it is a nonlocal and non-defining entity.
          attributes ← alter_attributes(attributes, link, nonlocal, non-defining);
          -- Modify this block to include the newly inherited object.
          add_new_info(this_block, identifier, kind, attributes);
        end loop;
      else
        -- The identifier is known to this_block; this is acceptable only under
        -- certain conditions, such as for function pseudo-variables.
        -- Gain access to both symbol tables.
        tab_info ← get_info_via_ident_from_sym_tab(parent, identifier);
        tab_info_2 ← get_info_via_ident_from_sym_tab(this_block, identifier);
        -- If both tables are the same size (i.e., 1), then they do not represent
        -- overloaded identifiers.
```

Figure 3.15. Scope rules for Pascal – the first pass.

```
          if and(equal(size_of_info_table(tab_info), 1),
                equal(size_of_info_table(tab_info_2), 1))
          then
            -- If the identifier represents a function defined in the parent
            -- block, and a function pseudo-variable defined in the current
            -- block, then we may inherit the overloaded identifier.
            if and(and(equal(get_kind_from_info_table(tab_info, 1), function),
                         defining_entry?(get_attributes_from_info_table(tab_info, 1))),
                     and(equal(get_kind_from_info_table(tab_info_2, 1),
                                function_pseudo_variable),
                         defining_entry?(get_attributes_from_info_table(tab_info_2, 1))))
            then
              -- Prepare the information to be inherited.
              attributes ← get_attributes_from_info_table(tab_info, 1);
              kind ← get_kind_from_info_table(tab_info, 1);
              -- Establish a RO link.
              link ← link_to(parent, identifier, kind, RO);
              -- Object inherited as a nonlocal, non-defining entity.
              attributes ← alter_attributes(attributes, link, nonlocal, non-defining);
              -- Modify this_block to include the newly inherited object.
              add_new_info(this_block, identifier, kind, attributes);
            end if;
          end if;
        end if;
      end loop;
    end if;
```

Figure 3.15. Continued.

If the identifier under consideration is already known to the current block (referred to as "this_block" in Figure 3.15), then it may only be inherited if the identifier represents a function pseudo-variable in the current block, whilst representing a defining entry for a function in the parent block. This is only possible if both the tables associated with the identifier in parent and this_block contain only a single entry. If this is so, then the attributes are modified and the link established. Otherwise, no action is taken; the identifier is simply not inherited.

Scope-Rules-Pass-2, shown in Figure 3.16, locates the parent block and considers each identifier in its symbol table in turn. If the identifier is unknown to the current

Scope-Rules-Pass-2 =

```
-- Find the nearest textually enclosing block; this is called the parent block.
locate_parent;
if not(empty_symbol_table?(parent))
then
  -- Iterate through the symbol table of the parent block.
  for i in 1 .. size_of_symbol_table(parent)
  loop
    -- Get an identifier.
    identifier ← get_ident_from_sym_tab(parent, i);
    -- Gain access to the attribute table associated with the identifier.
    tab_info ← get_info_from_sym_tab(parent, i);
    if not(member_of_symbol_table(this_block, identifier))
    then
      -- If the identifier is unknown to this_block then it may be inherited.
      -- Iterate over the attribute table.
      for j in 1 .. size_of_info_table(tab_info)
      loop
        attributes ← get_attributes_from_info_table(tab_info, j);
        -- It can be inherited only if it is declared nonlocal to the parent
        -- block. That is to say that it was inheritied by the parent block
        -- in the initial pass.
        if not(locally_declared(attributes))
        then
          -- Modify this block to include the newly inherited object.
          add_new_info(this_block, identifier,
                        get_kind_from_info_table(tab_info, j), attributes);
        end if;
      end loop;
    else
      -- The identifier was known to this_block; this is acceptable only under
      -- certain conditions, i.e., for functions and function pseudo-variables.
      -- Get the attribute table associated with the identifier.
      tab_info_2 ← get_info_via_ident_from_sym_tab(this_block, identifier);
```

Figure 3.16. Scope rules for Pascal – the second pass.

```
          -- If the size of the table is 1 (i.e., not overloaded)
          if equal(size_of_info_table(tab_info_2), 1)
          then
            -- If it defines a nonlocal function pseudo-variable in this_block.
            if and(equal(get_kind_from_info_table(tab_info_2, 1),
                         function_pseudo_variable),
                   not(locally_declared?(
                       get_attributes_from_info_table(tab_info_2, 1))))
            then
              -- If it defines a nonlocal function in the parent block.
              if and(member_of_info_table(tab_info, function),
                     not(locally_declared?(
                         get_attributes_from_info_table(tab_info, 1))))
              then
                -- Modify this_block to include the newly inherited object
                -- without alteration.
                add_new_info(this_block, identifier,
                             get_kind_from_info_table(tab_info, 1),
                             get_attributes_from_info_table(tab_info, 1))
              end if;
            end if;
          end if;
        end if;
      end loop;
    end if;
```

Figure 3.16. Continued.

block and is a nonlocal entity in the parent block, then it is inherited. In this case, the attributes do not need to be altered. If the identifier is known to this_block as a function pseudo-variable inherited as a result of Scope-Rules-Pass-1, and the identifier represents a nonlocal function in the parent block, then it can also be inherited.

3.3.5.3 Parameters

Pascal supports three distinct kinds of parameter transmission. These are variable, value, and functional and procedural parameters; the last two of these are sufficiently similar that they can be discussed together. The reader is referred to Figures 3.17,

```
Variable-Parameters(inst: activation_instance;
                    attrib: attribute_type;
                    act_param: actual_param_info) =

  -- If the actual parameter is an identifier.
  if represent_identifier?(act_param)
  then
    -- Get the parent activation instance.
    parent_activation ← get_parent(inst);
    -- Gain access to the attribute table associated with the identifier.
    tab_info ← get_info_tab_from_activation(parent_activation,
                                    return_name(act_param));
    -- If the attribute table size is 1, i.e., no overloading.
    if equal(size_of_info_tab(tab_info), 1)
    then
      attributes ← get_attributes_from_info_table(tab_info, 1);
      kind ← get_kind_i(tab_info, 1);
      -- If the kind of actual parameter is a variable, value-parameter
      -- or variable-parameter.
      if or(or(equal(kind, variable_parameter),
               equal(kind, value_parameter)),
            equal(kind, variable))
      then
        -- If it represents a defining entry.
        if defining_entry?(attributes)
        then
          -- If it is RW accessible.
          if read_write_accessible(attributes)
          then
            -- Establish a RW link.
            link ← link_dynamic(parent_activation,
                          return_name(act_param), kind, RW);
            attrib ← alter_attributes(attrib, link, local, non-defining);
            -- Update the activation instance.
            update_activation(inst, attrib);
          else
            Error("Violates principle of non-increasing privilege.");
          end if;
```

Figure 3.17. Variable parameters.

```
            else
              -- Make a link to the defining entry that the actual
              -- parameter is linked to.
              -- Ensure that the actual parameter is read-writeable.
              if and(read_write_accessible(attributes),
                    read_write_accessible(return_link(attributes)))
              then
                -- Establish link
                link ← return_link(attributes);
                attrib ← alter_attributes(attrib, link, local, non-defining);
                -- Update the activation instance.
                update_activation(inst, attrib);
              else
                Error("Cannot violate principle of non-increasing privilege.");
              end if;
            end if;
          else
            Error("Actual parameter is of an inappropriate kind.");
          end if;
        else
          Error("Variable formal parameter is incompatible with a function
                  or function pseudo variable.");
        end if;
      else
        Error("Actual parameter is expected to be an identifier.");
      end if;
```

Figure 3.17. Continued.

3.18 and 3.19 for the definitions of the relevant semantic routines; these definitions highlight the essential differences between the transmission modes.

Each routine takes three parameters – an activation instance representing the executing routine in which the formal parameters reside, the attributes of the formal parameter being considered at present and the matching actual parameter. The principal difference between the handling of parameters and the semantics of scope rules and local declarations is that parameters deal with the dynamic (or run-time) environment, as the formal parameter (which was handled initially by the routine for

```
Value-Parameters(inst: activation_instance;
                 attrib: attribute_type;
                 act_param: actual_param_info) =

  -- If the actual parameter is an identifier.
  if represent_identifier?(act_param)
  then
    -- Gain access to the parent activation instance (i.e., the caller).
    parent_activation ← get_parent(inst);
    -- Gain access to the relevant attribute table.
    tab_info ← get_info_tab_from_activation(parent_activation,
                                            return_name(act_param));
    -- If the size of the attribute table is 1, i.e., no overloading.
    if equal(size_of_info_tab(tab_info), 1)
    then
      -- Gain access to the attributes.
      attributes ← get_attributes_from_info_table(tab_info, 1);
      kind ← get_kind_i(tab_info, 1);
      -- Actual parameter must be a variable, value-parameter or
      -- variable parameter.
      if or(or(equal(kind, variable_parameter),
               equal(kind, value_parameter)),
            equal(kind, variable))
      then
        -- If it is a defining entry.
        if defining_entry?(attributes)
        then
          -- If it is read-only or read-writeable.
          if or(read_write_accessible(attributes),
                read_only_accessible(attributes))
          then
            -- Establish a RO link.
            link ← link_dynamic(parent_activation, return_name(act_param),
                                kind, RO);
            attrib ← alter_attributes(attrib, link, local, defining);
            -- Update the activation instance.
            update_activation(inst, attrib);
          else
            Error("Violates principle of non-increasing privilege.");
          end if;
```

Figure 3.18. Value parameters.

```
          else
            -- Make a link to the defining entry to which the actual parameter
            -- is linked. Ensure object is RW or RO accessable.
            if and(or(read_write_accessible(attributes),
                          read_only_accessible(attributes)),
                     or(read_write_accessible(return_link(attributes)),
                          read_only_accessible(return_link(attributes))))
            then
                     -- Establish link.
                     link ← return_link(attributes);
                     attrib ← alter_attributes(attrib, link, local, defining);
                     update_activation(inst, attrib);
            else
                     Error("Cannot violate principle of non-increasing privilege.");
            end if;
          end if;
        else
          Error("Actual parameter is of an inappropriate kind.");
        end if;
      else
        Error("Actual parameter may not be a function or
                         function pseudo-variable.");
      end if;
    else
      -- Actual parameter was a value, so its value can be stored.
      attrib ← store(attrib, value_of(act_param));
      update_activation(inst, attrib);
    end if;
```

Figure 3.18. Continued.

```
Proc-And-Func-Parameters(inst: activation_instance;
                         attrib: attribute_type;
                         act_param: actual_param_info) =

 -- If the actual parameter is an identifier.
 if represent_identifier?(act_param)
 then
   -- Get the activation instance of the calling block.
   parent_activation ← get_parent(inst);
   tab_info ← get_info_tab_from_activation(parent_activation,
                                   return_name(act_param));
   -- Check that the identifier represents a function or a procedure
   -- in the calling block.
   if member_of_info_table(tab_info, function)
   then
     kind ← function;
   else
     if member_of_info_table(tab_info, procedure)
     then
       kind ← procedure;
     end if;
   end if;
   -- Get the attributes associated with the identifier.
   attributes ← get_attributes_via_kind_from_info_table(tab_info, kind);
   -- Check that the actual and formal procedural or functional
   -- parameters are compatible.
   if match_param_info(attrib, attributes)
   then
     -- Ensure it is a defining entry.
     if defining_entry?(attributes)
     then
       -- Ensure it is Read-Only accessible.
       if read_only_accessible(attributes)
       then
         -- Establish the link.
         link ← link_dynamic(parent_activation,
                      return_name(act_param), kind, RO));
         attrib ← alter_attributes(attrib, link, local, non-defining);
         update_activation(inst, attrib);
       else
         Error("Violates principle of non-increasing privilege.");
       end if;
```

Figure 3.19. Procedural and functional parameters.

```
        else
          -- Make a link to the defining entry that the actual parameter
          -- is linked to. Ensure it is RO accessable.
          if and(read_only_accessible(attributes),
                  read_only_accessible(return_link(attributes)))
          then
            -- Establish the link.
            link ← return_link(attributes);
            attrib ← alter_attributes(attrib, link, local, non-defining);
            update_activation(inst, attrib);
          else
            Error("Cannot violate principle of non-increasing privilege.");
          end if;
        end if;
      else
        Error("Actual and formal parameters are not compatible.");
      end if;
    else
      Error("Actual parameter is expected to be an identifier.");
    end if;
```

Figure 3.19. Continued.

local declarations) can be linked to several, possibly different, actual parameters over the lifetime of the program.

Parameters, described in Figures 3.17, 3.18 and 3.19, are handled by first identifying the transmission mode of the actual parameter. This determines the course of action to be taken. If it is appropriate, the calling activation instance containing the actual parameter is then located. Next, the actual parameter is retrieved and checked to ensure that it is of the appropriate kind. If it represents a defining entry and the actual parameter is a RW object in the parent activation instance, then the formal parameter is linked to the actual parameter. If the actual parameter in the parent activation instance is a non-defining entry, then the link leaving this is followed and the formal parameter is linked to the resulting defining entry.

The descriptions in these figures each issue error messages indicating that the "principle of non-increasing privilege" has been violated if an attempt is made to gain access to an object through a parameter in such a way that the called procedure or function has greater access to the object than the original calling block. This principle was described in [72] and the term was coined in [83]. It is a principle which Pascal adheres to, in that a block cannot have read-only access to an object and then invoke a procedure or function and allow the newly activated routine to have read-write access to the object.

3.4 Observations

The most common way of defining the semantics of a programming language is that of a natural language description. As is well known, this approach suffers from many problems – ambiguities, omitted details, poorly defined aspects of the language concerned, and so on. In order to reduce the risk and severity of these difficulties, natural language descriptions have become more precise (at the cost of readability), but even this has not solved the problem completely.

The operational semantic model produced by the multi-layer technique described above is simpler to understand and read than, say, the semantic description of Pascal in terms of attribute grammars given by Kastens *et al.* [68]. An operational model is also capable of describing both the static and dynamic semantics of a language, whilst attribute grammars tend to perform well in a description of the static semantics only. At present, the British Standards Institute is attempting to define the semantics of Modula-2 using VDM [13]; however, VDM is itself currently being standardized and will suffer the problems typical of any programming language standard. Axioms used in an axiomatic approach do not suffer any such problem, as they have a firm mathematical foundation.

The use of ADT's in the description of the data structures used in the model lead to the development of a layered model that caters for programmers, compiler writers and language designers. Even though each group requires a different depth of understanding of the language, it is now possible to produce one document to satisfy all of these groups, rather than having to write several documents, each aimed at a different group.

A novel approach to the use of algebraic techniques in the definition of a programming language has been demonstrated. These algebraic techniques provide the degree of formalism necessary to establish a suitable base from which to build a precise model.

Chapter 4

A Tool for Language Definition and Interpreter Synthesis

4.1 Introduction

The previous chapter introduced an operational semantic model for the definition of aspects of the semantics of programming languages; this model was illustrated by using it to describe data control in the Pascal programming language. This exercise served to highlight those aspects of the semantics of Pascal which are awkward to describe and that are difficult for users of the language to learn and grasp; these aspects include function value return versus function call, scope rules, and the various parameter transmission modes. The model clearly demonstrates the difficulties in the semantics of these features and this suggests that such a model is a suitable vehicle for the definition of programming languages and a useful tool to aid in their design.

These claims are not new; they are in fact applicable to most formal techniques. However, as discussed in Section 1.2, none of the existing formalisms have overwhelming support across the entire computer science community. There are many reasons for this, but one contributing factor is the inability to generate an implementation from the definition of a programming language using many of these techniques. Efforts are currently underway to rectify this for formal techniques such as denotational semantics [73, 146] and attribute grammars [31, 68, 75, 140], as well as through tools such as that provided by the Amsterdam Compiler Kit [69, 70, 133, 136, 137, 142].

It is important that language designers be encouraged to formally define new programming languages from the outset rather than employing the common practice of informally specifying the language, implementing various compilers (with slightly different behaviour with regard to semantics) and waiting for the language to gain wide acceptance and usage before attempting a formal definition. This course of action has been observed time and time again with the development of languages such as Pascal [65, 155], C [71] and Modula-2 [156, 157].

The U.S. Department of Defense addressed this issue by demanding a complete language definition for Ada [141] before any compilers were built. This ensured (with the help of a validation suite) that the compilers which were eventually produced behaved in a reasonably similar manner to each other (although they have still suffered from inconsistencies). The difficulty of this approach has been that no implementation could be generated directly from the definition, and hence there has been no reference implementation against which to compare the results of a hand-crafted compiler. Along with the fact that a potentially ambiguous description exists (see Chapter 5), and that compilers are validated empirically, it is interesting to notice that the production of the definition prior to the production of a compiler has resulted in a more uniform collection of compilers than has been achieved for many earlier programming languages. This is a clear indication that there is merit in providing a rigid definition before an implementation is produced.

Language designers require some incentive to produce a formal definition of a new programming language. It is all too easy to follow the traditional path and merely generate an informal description, allowing future implementors to guess, or assume, intentions which are not apparent in the description. This incentive may take many forms, one of which is the generation of an implementation directly from the formal programming language definition. The operational model of the previous chapter is amenable to such an approach. The advantage is that the language designer is provided with an implementation with which to experiment and verify the specified semantics of the programming language, and users are presented with a clear, unambiguous definition. Compiler writers also benefit in that they obtain a reference implementation against which to compare the results of their own work.

ATLANTIS, A Tool for LANguage definiTion and Interpreter Synthesis, is based on the operational model of Chapter 3 and provides a mechanism for the formal definition of a programming language and the generation of an interpreter from it. The model has identified two aspects of the language definition, namely syntax and semantics, and ATLANTIS introduces a third – the lexical component. Each of these components of language definition used by ATLANTIS are discussed in the following sections.

The ATLANTIS system has been implemented and has been tested using a small programming language derived from Pascal. This small language, called Neptune, was derived from Pascal by omitting several features and by introducing some new features. The features removed from Pascal include: labels, constants, type definitions, arrays, records, the character type, variable parameters, with statements, goto statements, repeat loops, and trigonometric operations. The following features not found in Pascal are also present in Neptune: strings, a general loop construct, an exit statement, and a return statement. Each of the added features bears a great deal of similarity to the corresponding construct in the Ada programming language.

The language Neptune, despite the omission of several features of Pascal, is still a non-trivial language which embodies many of the semantic complexities of Pascal, such as scope rules and some aspects of parameter transmission. The Neptune definition occupies a total of 4547 lines and is the source of several of the examples presented in this chapter. Further details of the Neptune definition can be found throughout this chapter and in Appendices C and D. Appendix C provides an outline of the ADT's used in the Neptune definition and in Appendix D provides an outline of the HLO's used.

4.2 General

ATLANTIS mimics the layered operational model introduced in the previous chapter. As with the operational model, users may read the ATLANTIS definition of the programming language to the depth which best suits their needs before leaving the formal definition and referring to the informal narrative which accompanies it. This informal narrative is provided to ATLANTIS by introducing comments into the programming language definition. These comments do not form part of the formal language definition, but provide a description of the various layers within the model which together form the formal definition. Comments may be used anywhere within an ATLANTIS definition; a comment in ATLANTIS commences with the symbol "--" and continues until the end of the line.

ATLANTIS differs from other tools which generate portions of a compiler from a formal description, such as GAG [68, 140] and the Amsterdam Compiler Kit [69, 70, 133, 136, 137, 142], and which attempt to generate a complete compiler. In contrast, ATLANTIS generates an interpreter. The reason for making no attempt to produce machine code is that the generation of machine code is specific to a particular architecture, and not to the language. In other words, it is not part of the language definition to specify what machine code is to be produced by which language construct; it is the job of the language definition to define how that construct is to behave and nothing more. However, it may be possible to separate out the hardware-specific details and generate a true compiler, but this is likely to rely on a particular intermediate representation, in order to ensure that the generated compiler could be ported to various machines. The generation of an interpreter obviates the need for any particular intermediate representation and is, in fact, sufficient as a reference implementation.

4.3 Lexical Analysis

One of the most fundamental aspects of a programming language definition involves the specification of the lexical components of the language. These lexical components are the building blocks from which working programs in the new language are to be built. ATLANTIS recognizes that language designers may wish to alter a lexical element once language design is underway; for example, the designer may wish to change a keyword from "start" to "begin". This alteration can be done simply by varying the association between a keyword and a symbol used as a reference to this lexical token throughout the remainder of the language definition.

A language definition in ATLANTIS commences with a statement which specifies the name of the language and whether or not the language is case independent. A statement such as:

Language example **is case dependent**

introduces a language called "example" in which the case of characters plays an important role in the distinction between identifiers; hence, for example, "BeGiN" and "bEgIn" are treated as completely different identifiers. A statement such as:

Language example2 **is case independent**

introduces a language, named "example2", in which the case of characters plays no special role. In this instance, the identifiers "BeGiN" and "bEgIn" are indistinguishable.

ATLANTIS itself is case independent and only takes note of the case used to specify keywords in the defined language if the language is deemed to be case dependent. The bold face words in the examples thus far represent keywords in ATLANTIS. If any of these keywords are required as part of the definition of a new programming language, then ATLANTIS can be convinced to treat them as ordinary strings by prefixing the keyword by a backward slash ("\"). Hence "\CASE" represents the string "CASE" rather than the keyword "**case**".

Following the introduction of the language name and the specification of its case (in)dependence, the lexical tokens of the new language are defined. The **keyword** section of an ATLANTIS definition specifies a symbol which is bound to a lexical token; the keyword-symbol binding is terminated by a semicolon. Whitespace is unimportant unless it is made explicit by representing each blank character by a backward slash ("\") followed by a space. The lexical token is not directly referenced outside the keywords section, instead the associated symbol is used throughout the remainder of the definition.

The lexical tokens listed in the keywords section include reserved words and operators. In fact, all lexical tokens which are to be referenced in the syntactic definition of the language are defined in the keywords section. Each token must have a unique binding to a symbol and symbols may not be overloaded. As mentioned earlier, ATLANTIS itself is completely case independent except if the programming language being defined is specified as being case dependent. In this situation, the case of lexical tokens defined in the **keyword** section is significant and this information is bound to the corresponding symbol.

An example of a typical keywords section, taken from the Neptune definition, is given in Figure 4.1. From this figure, we see the use of the backward slash to alter the meaning of a special character sequence to be simply a string. For example, to refer to the semicolon character rather than terminating the association, the token is entered as "\;". The only other strings which might need to be prefixed by a backward slash are the keywords "**operator**", "**special**", "**definition**" and "**model**" (all of which introduce other sections of the ATLANTIS definition), the comment symbol "--" and the backward slash character itself. A backward slash preceding any other character simply returns the character itself except within the specials section (discussed shortly). This means that a backward slash preceding the space character simply returns the space character.

After the specification of the language keywords and the corresponding symbol to represent that keyword throughout the remainder to the language definition, the operators of the language may be further defined. Operators may have their precedence within expressions specified, as well as their associativity. ATLANTIS treats the language operators as keywords and, as a result, the matching symbol is used when specifying the priority and precedence of the symbol.

```
Keyword
    -- Symbol            Token
    and_sym              and ;
    begin_sym            begin ;
    boolean_sym          boolean ;
    call_sym             call ;
    else_sym             else ;
    elsif_sym            elsif ;
    end_sym              end ;
    exit_sym             exit ;
    false_sym            false ;
    float_sym            float ;
    function_sym         function ;
    if_sym               if ;
    input_sym            input ;
    integer_sym          integer ;
    loop_sym             loop ;
    not_sym              not ;
    null_sym             null ;
    or_sym               or ;
    output_sym           output ;
    procedure_sym        procedure ;
    program_sym          program ;
    return_sym           return ;
    string_sym           string ;
    then_sym             then ;
    true_sym             true ;
    var_sym              var ;
    when_sym             when ;
    while_sym            while ;
    becomes_sym          := ;
    comma_sym            , ;
    semicolon_sym        \; ;
```

Figure 4.1. The keyword section of an ATLANTIS definition.

colon_sym	: ;
left_bracket	(;
right_bracket	) ;
period_sym	. ;
equal_sym	= ;
not_equal_sym	# ;
less_than_sym	< ;
less_or_equal_sym	<= ;
greater_than_sym	> ;
greater_or_equal_sym	>= ;
plus_sym	+ ;
minus_sym	− ;
mult_sym	* ;
divide_sym	/ ;

Figure 4.1. Continued.

The precedence of an operator is given as a positive number. The larger the number, the greater the priority. If two operators are given the same priority and an expression in the new language involves both operators, then the operator which is listed first in the ATLANTIS definition is not necessarily evaluated first. ATLANTIS does not attempt to specify the order of evaluation in this case; rather, it is left to the language designer to cover this question as part of the details of the programming language definition. If it is left unspecified, then the implementation is free to choose the order of evaluation in a nondeterministic fashion.

The associativity given to an operator may be any of "**left**", "**right**" or "**nonassoc**", which specify that the operator is left associative, right associative or nonassociative in nature, respectively. The operator specification part of the ATLANTIS definition for the Neptune language is given in Figure 4.2.

Finally, ATLANTIS recognizes that programming languages usually have several lexical elements which have a specific structure and consist of a specific set of characters in a certain order, such as "BEGIN", and these are also handled in the **keyword** section of the language definition. Most languages also have lexical components which

```
operator
   equal_sym              1   nonassoc ;
   not_equal_sym          1   nonassoc ;
   less_than_sym          1   nonassoc ;
   less_or_equal_sym      1   nonassoc ;
   greater_than_sym       1   nonassoc ;
   greater_or_equal_sym   1   nonassoc ;
   plus_sym               2   right ;
   minus_sym              2   right ;
   mult_sym               3   right ;
   divide_sym             3   right ;
   or_sym                 5   right ;
   and_sym                6   right ;
   not_sym                9   right ;
```

Figure 4.2. The operator specification within the definition of Neptune.

do not have such a rigid structure, but rather they simply have a general form to which they comply. Examples of such lexical elements include identifiers, strings and comments.

An identifier, for example, is an important lexical component that may appear in certain locations within a valid program written in the defined language. It is important for the language designer to be able to specify the nature of an identifier as a lexical component. To do this, ATLANTIS introduces a **special** section to specify these special lexical tokens. These tokens are special in that they have a structure which may match a variety of input strings. If their structure overlaps with a language token defined in the **keyword** section, then the token specified in the **keyword** section is always found in preference to the token from the **special** section. This prevents a keyword such as "BEGIN" from being recognized as an identifier rather than the language keyword that it is.

To aid in the specification of the structure of these special lexical tokens, ATLANTIS introduces certain symbols to match a class of characters known as a syntactic

Symbol	Meaning
\l	lower case letter (a–z).
\u	upper case letter (A–Z).
\d	any digit (0–9).
\c	any printable character except the space character, tab and newline.
\a	any alphanumeric character (a–z, A–Z, 0–9).
\b	the space character.
\n	the newline character.
\t	the horizontal tab character.

Table 4.1. The syntactic categories for the definition of special lexical elements.

EBNF	Meaning
(...)	is used for grouping items together.
...|...	represents a choice between items.
{...}	represents zero or more instances of the items between the braces.
{...}+	represents one or more instances of the items between the braces.
{...}*n*	where *n* is any positive number, represents exactly *n* occurrences of the items between the braces.
[...]	represents an optional item.

Table 4.2. EBNF symbols used in ATLANTIS.

category. These syntactic categories are listed in Table 4.1. The structure of the special lexical tokens is specified via the EBNF-like notation shown in Table 4.2

Using the categories in Table 4.1 and the notation in Table 4.2, the language designer is able to define the nature of lexical components such as identifiers, strings and comments. An example of the definition of such special tokens within an ATLANTIS definition is given in Figure 4.3; this is taken from the definition of the Neptune language.

Lexical elements defined in the **special** section, such as those shown in Figure 4.3, may be supersets of lexical tokens previously specified in the **keyword** section of the

```
special
   ident       \a{(\a|\d)} ;
   integer     \d{\d} ;
   float       \d{\d}.\d{\d} ;
   string      "{(\c|\b)}" ;
   comment     \-\- {(\c|\b)}\n ;
```

Figure 4.3. The special token definitions within an ATLANTIS definition.

ATLANTIS definition. In these cases, ATLANTIS always tries to match the keyword before attempting to match a special token.

ATLANTIS uses the special symbol "comment" as the symbolic name of a lexical element which is to be ignored by the generated interpreter. In the example provided in Figure 4.3, a form of a comment in the programming language is specified as commencing with "--" followed by any number of characters and spaces, and finally terminated by the end of the line. The generated interpreter based on this definition recognizes this pattern as valid and simply ignores it. In this way, comments may appear anywhere in the newly defined language, provided they take the form specified. The remainder of the language definition need not concern itself further with comments. If the language designer wishes to restrict the use of comments in the language being defined, then a name other than "comment" must be chosen and each location where a comment may validly appear in a program written in the new language must be specified as part of the language definition.

The three sections described above – **keyword**, **operator** and **special** – are sufficient to specify the lexical components of the programming language being defined and even allows small amounts of semantic detail to be specified, such as precedence and associativity of operators. This semantic information simply makes the language

UNIVERSITY OF ADELAIDE LIBRARY

definition easier to produce, as it is no longer necessary to represent operator precedence via the syntactic definition of the language. The result is a language definition which is easier to produce and easier to read than many other formalisms.

Salomon and Cormack [123] point out that many language definitions do not specify the form, or position, of allowed whitespace. The reason is that a definition which specifies the allowable locations of whitespace is cluttered and, as a consequence, difficult to read. However, if a language designer wishes to specify the form and permitted locations of whitespace then the designer may do so via the special symbols representing whitespace in ATLANTIS (such as \n, \b, etc). To specify the whitespace component of a programming language at this level of detail is to lengthen the language definition considerably for dubious reward in many circumstances. However, the language designer is free to follow the advice of Rose *et al.* [121] and specify the formatting as part of the syntactic definition of the programming language being defined. ATLANTIS assumes, by default, that whitespace such as blank characters, horizontal tabs, carriage returns, etc., may occur between any pair of lexical tokens; in other words, free formatting is the default option within the ATLANTIS system.

4.4 The Syntactic Component

Programming language syntax is traditionally specified with BNF or syntax charts (railroad diagrams), as discussed in Section 1.1. Techniques such as BNF adequately describe programming language syntax, and are well known and well understood. As such, there is little need to tamper with the technique, other than employ an enhancement in the form of extended BNF (EBNF) which simply provides a shorthand notation to reduce the length of a definition. The EBNF notation used in ATLANTIS

```
block:  [ label_declaration_part ]
        [ constant_definition_part ]
        [ type_definition_part ]
        [ variable_declaration_part ]
        [ procedure_and_function_declaration_part ]
        statement_part
        ;
```

Figure 4.4. The syntactic definition of a Pascal block.

to specify the syntax of a programming language is identical to that introduced for the specification of the special lexical tokens (see Table 4.2).

Using this notation, Figure 4.4 provides the syntactic definition of a Pascal block. The various components which comprise a block are specified and their ordering is determined. As can be seen from the figure, the majority of the components which comprise a Pascal block are optional, with only the "statement_part" being mandatory. The semicolon which appears at the end of the syntactic definition denotes the completion of the definition.

The definition of the syntactic components of a programming language in ATLANTIS is little different to a typical EBNF description of the syntactic components of a programming language found in many natural language definitions. As such, this aspect of ATLANTIS introduces no new concepts.

Identifiers used in the syntactic definition of the programming language may be nonterminal symbols which are specified elsewhere in the definition, or terminal symbols as defined in the lexical portion of the definition (**keyword** and **special** sections).

The syntactic description of the new programming language may be defined in any order. There is no need to define a syntactic category (nonterminal symbol) before it may be used. ATLANTIS constructs a table of all names used and defined within the language definition, performing appropriate checking once the entire syntactic definition is provided to ensure that all nonterminal symbols used have been properly defined. There is no compulsion to use all of the terminal symbols which have been defined.

4.5 The Semantic Component

ATLANTIS is based on the operational semantic model discussed in the previous chapter. As a result, implementation of a semantic description of a new programming language will involve the building and manipulation of information structures, as well as multiple passes over some representation of the source code of a program written in the language. The model of the previous chapter is also layered and this characteristic manifests itself within the ATLANTIS system.

The outermost layer of the model provides the most abstract definition of the programming language semantics and, in ATLANTIS, this layer of detail is tied directly to the syntactic definition of the programming language. This outermost layer makes calls on various semantic routines defined in an inner layer of the model, defining the semantic actions which are to take place when certain syntactic entities are encountered during the various passes over the source program. This linking of syntax and semantics means that it is particularly simple for a reader to locate a semantic routine in order to answer most language questions. In the majority of cases, the reader will know the syntactic form which is related to the area of difficulty and hence can find the name of the desired semantic routine easily.

The outermost layer of the model is represented by the distribution of the calls to semantic routines (HLO's) throughout the syntactic definition of the language. The syntactic definition and the semantic routines are distinguished from each other by

```
block : %% Pass 2:
             call Scope_Rules_Pass_2;
        %%
        [label_declaration_part]
        [constant_definition_part]
        [type_definition_part]
        [variable_declaration_part]
        [procedure_and_function_declaration_part]
        %% Pass 1:
             call Scope_Rules_Pass_1;
        %%
        statement_part
        ;
```

Figure 4.5. ATLANTIS definition of a Pascal block with semantic definitions.

delimiting the semantic regions (regions containing calls to semantic routines) by the symbols "%%".

If the language requires multiple passes to describe its semantics, then the pass on which the semantic action is to have an effect is specified, using the notation "**Pass** n: ...". All actions within a semantic region are evaluated during the specified pass. If no particular pass is specified, then the semantic region is assumed to be applicable in the first pass. This approach makes it explicit as to which actions are to take place at which time and in conjunction with the detection of which syntactic structure. For any particular semantic region, the pass over the source file in which it is to have an effect can only be specified once, and it must be specified before any semantic actions are detailed within that semantic region. Hence, a Pascal block (which was described previously in the underlying model in Figure 3.7) is described to ATLANTIS as shown in Figure 4.5. The **call** constructs in Figure 4.5 indicate calls to semantic actions.

All semantic regions are linked to the preceding syntactic structure, if there is one. If no syntactic structure is present, or suitable, then a null syntactic node is

```
identifier : %% call routine_1;
             %%
             letter
                  { [ underscore
                        %% call routine_2;
                        %%
                    ]
                    %% call routine_3;
                    %%
                    (digit
                    | letter
                        %% call routine_4;
                        %%
                    )
                    %% call routine_5;
                    %%
                  }
                  %% call routine_6;
                  %%
             ;
```

Figure 4.6. An identifier definition for ATLANTIS.

created. The semantic regions are only evaluated when the relevant syntactic structure is recognized and the current pass over the source program corresponds to that specified in the semantic region. The syntactic structure to which a semantic region may be linked may be either a terminal or a nonterminal symbol in the language being defined. As a result, the EBNF notation used for syntactic definition also affects the location and evaluation of the semantic regions. This is illustrated in the exaggerated syntactic and semantic definition of an identifier given in Figure 4.6.

Figure 4.6 indicates that the semantic routine "routine_1" must always be evaluated before the identifier is elaborated any further. This semantic action is bound to a null syntactic node which was specifically created since there was no preceding syntactic entity. Syntactically, a letter must next be present (and will be consumed by the generated scanner), and this may be followed by zero or more occurrences of the specified tokens. If any of the specified tokens are found, there are certain semantic actions which must be evaluated depending on the token. If the optional underscore token is detected, then the semantic action "routine_2" is invoked; it is not evaluated if an underscore is not located by the scanner. The optional underscore token is followed by a call to the semantic action "routine_3" which is unconditional if the next character in the input stream is a letter or a digit. After evaluation of this semantic routine, the next token is examined. If it is a letter, then it is accepted and the semantic action "routine_4" is evaluated; if it is a digit then the token is accepted but no semantic action is evaluated. After handling the choice between digit and letter, the semantic action "routine_5" is unconditionally evaluated. If the next character is a letter, digit or underscore then the contents of the repetitive construct is repeated, and hence calls to some of the semantic routines may occur multiple times as dictated by the input stream. When the identifier is completely recognized syntactically, the semantic action "routine_6" is unconditionally evaluated.

This EBNF syntactic description with associated semantic calls shown in Figure 4.6 can be represented in the flowchart shown in Figure 4.7 which clearly demonstrates the points at which semantic routines are to have an effect.

The semantic regions within the syntactic definition of the language have access to the environment variables used by the language definition. Environment variables are a mechanism which allow the author of a language definition to refer to information structures, or components of them, by name; use of these variables can improve the readability of the language definition. Environment variables are described in detail in Section 4.7; however, it is worth noting here that their values may be altered within the semantic regions via direct assignment, or through calls to high level operations or

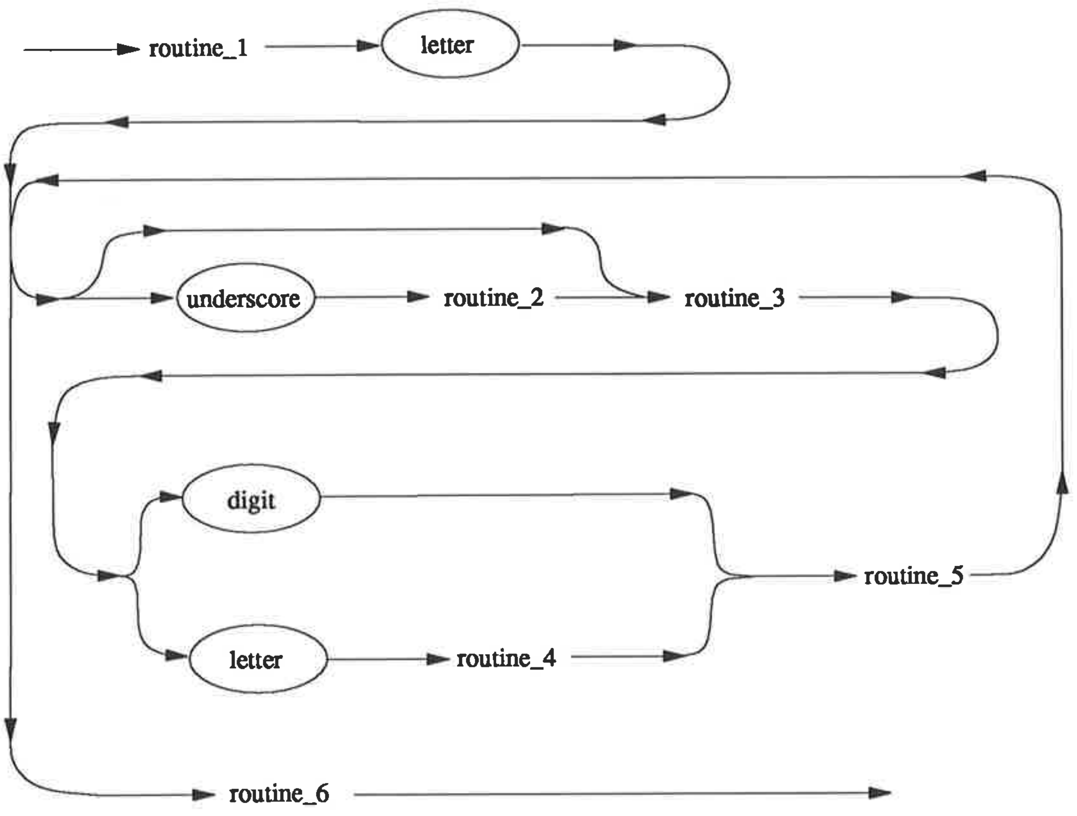


Figure 4.7. Flow chart illustrating syntax and semantics.

abstract data type operations. Furthermore, the semantic actions referenced within a semantic region can be ordered through sequencing with the ";" operator, selection between actions through the **if–then** and **if–then–else** constructs, and iteration using the various loop constructs to be discussed in Section 4.8.

As already mentioned, the semantic definition of a programming language within the ATLANTIS framework follows the model introduced in the previous chapter. Hence, there are additional layers of the definition which must be considered; specifically, the ADT specifications and the HLO definitions have yet to be addressed. These aspects are defined in the following sections. ATLANTIS introduces the elaboration of the semantic details pertaining to the language definition by indicating the end of the syntactic definition and the start of the semantic definition with the keyword "**model**".

4.6 Abstract Data Type Definitions

The description of ATLANTIS has thus far covered the specification of the lexical components of the programming language, followed by the syntactic definition annotated with semantic actions corresponding to the outermost layer of the semantic model. The next part of the input to ATLANTIS defines the ADT's used in the model; in other words, the innermost layer of the model is defined next. This allows ATLANTIS to behave as a single pass system and permits a symbol table to be built which allows the usage of the ADT's and HLO's to be checked at all levels. From this point onwards in the language definition, ATLANTIS insists on the declaration of all identifiers before they are used.

As discussed earlier in Chapter 3, the ADT's form the building blocks of the information structures used to describe the language semantics. An example of an ADT specified for ATLANTIS is given in Figure 4.8. From this figure, it can be seen that an ADT definition within the underlying model is introduced by the keyword "**ADT**" which is followed by an identifier which becomes the name of the data type defined by the ADT. After the ADT name, there is an optional list of identifiers, delimited by brackets, which denote types to be passed to the ADT later. The example in Figure 4.8 has two type parameters, "index_type" and "element_type", which will be specified when the ADT is instantiated. Parameterized ADT's allow the specification of generic or polymorphic ADT definitions, avoiding the need to specify a table for each index/element pair. This allows the language designer to concentrate on the language specification, rather than having to be overly concerned with housekeeping details. Parameterized ADT's also serve to keep the length of a language definition to a minimum.

```
ADT table [index_type, element_type];

  sorts table/index_type, element_type, boolean, integer;

  where index_type has equal:       index_type * index_type -> boolean;
        integer has    add_integer: integer * integer       -> integer;

  syntax
    new_table:                                        -> table;
    empty_table:   table                              -> boolean;
    member_table:  table * index_type                 -> boolean;
    insert_table:  table * index_type * element_type  -> table;
    remove_table:  table * index_type                 -> table + {error};
    alter_table:   table * index_type * element_type  -> table + {error};
    search_table:  table * index_type                 -> element_type + {error};
    size_of_table: table                              -> integer;

  semantics
    declare
      tab: table;
      index_1, index_2: index_type;
      elem_1, elem_2: element_type;

  axioms
    (1)  empty_table(new_table) = true;
    (2)  empty_table(insert_table(tab, index_1, elem_1)) = false;
    (3)  member_table(insert_table(tab, index_1, elem_1), index_2) =
           if equal(index_1, index_2)
           then true;
           else false;
           fi;
    (4)  member_table(new_table, index_1) = false;
    (5)  remove_table(insert_table(tab, index_1, elem_1), index_2) =
           if equal(index_1, index_2)
           then tab;
           else insert_table(remove_table(tab, index_2), index_1, elem_1);
           fi;
```

Figure 4.8. The definition of a table for input to ATLANTIS.

```
(6)  search_table(insert_table(tab, index_1, elem_1), index_2) =
        if equal(index_1, index_2)
        then elem_1;
        else search_table(tab, index_2);
        fi;
(7)  alter_table(tab, index_1, elem_1) =
        insert_table(remove_table(tab, index_1), index_1, elem_1);
(8)  insert_table(insert_table(tab, index_1, elem_1), index_2, elem_2) =
        if member_table(insert_table(tab, index_1, elem_1), index_2)
        then alter_table(insert_table(tab, index_1, elem_1), index_2, elem_2);
        else insert_table(insert_table(tab, index_2, elem_2), index_1, elem_1);
        fi;
(9)  size_of_table(new_table) = 0;
(10) size_of_table(insert_table(tab, index_1, elem_1)) =
        add_integer(1, size_of_table(tab));
```

Figure 4.8. Continued.

The ADT definition, whether parameterized or not, is then followed by a **sorts** clause which specifies two things. To the left of the "/" character is the name of the ADT that is being defined by the ADT definition. This name should be the same as that specified in the ADT header. After the "/" character is a complete list of all other data types used in the ADT definition. This list includes any type parameters. Only the types specified in this list may be referenced by the remainder of the ADT definition. An attempt to use any data type not specified within the **sorts** clause will result in ATLANTIS reporting an error. The **sorts** clause also provides a convenient mechanism to indicate the reliance of the ADT on other data types, allowing readers to quickly see the relationship.

An optional **where** clause follows the **sorts** clause. The **where** clause lists some operations assumed for data types listed in the **sorts** clause. In each case, the name of the operation, the relevant arguments and the result type of the operation is given. Figure 4.8 has a **where** clause which reads:

```
where index_type has equal:       index_type * index_type -> boolean;
      integer has    add_integer: integer * integer       -> integer;
```

This informs the reader, and the ATLANTIS system, that the type parameter which eventually corresponds to "index_type" must have an operation called "equal" which takes two objects of the type corresponding to "index_type" as arguments and returns a boolean result. This property will be checked by the ATLANTIS system when the ADT is instantiated with the appropriate actual parameters.

The ADT definition also assumes that the type integer provides an operation called "add_integer" which takes two integer arguments and returns an integer result. The axioms specifying the behaviour of the ADT can assume the existence of the operations listed in the **where** clause and hence employ them. ATLANTIS checks the existence of all operators listed in the **where** clause by examining the appropriate type at the appropriate time. This is performed immediately, if possible, as is the case with the type integer; otherwise, it is done at the time of instantiation (in the case of a type parameter, for example). Operations provided by other types, but not specified in the **where** clause, remain inaccessible to the ADT being defined and any attempt to refer to such operations are flagged as errors by the ATLANTIS system.

The keyword "**syntax**" introduces the specifications of the operations to be defined for the ADT. For each operation, its name, the types of each of its arguments and the result type are listed. From Figure 4.8, we find the specification of the operation "alter_table" given as:

```
alter_table:    table * index_type * element_type -> table + {error};
```

which introduces the operator "alter_table" and defines it such that every call to the operation requires three arguments, the first of which is of type "table", the second of type "index_type" and the third of type "element_type". The operation is to return a "table" object or an error result, as indicated by the presence of "+ {error}" in the result type given after the arrow symbol "->". The specification for each operator is terminated by a semicolon. The use of the semicolon is simply to provide some redundant information to the ATLANTIS system to improve error detection and handling. The section describing the syntax of ADT operators in ATLANTIS is almost identical to the style of ADT definitions presented in Chapter 2.

The final section of the ADT definition provides the semantics of the operations. This section is introduced by the keyword "**semantics**". The semantic definition is divided into two subsections; the first of which is introduced by the keyword "**declare**", and the second of which is introduced by the keyword "**axioms**".

The first subsection is optional and only serves to introduce metavariables to represent arbitrary objects of specified types in the subsequent subsection which provides the axiomatic definition of the operations. For example, the following is extracted from Figure 4.8:

index_1, index_2: index_type;

This declaration introduces the identifiers "index_1" and "index_2" to the ADT definition. These names can then be used in the axiomatic definition of the operations to represent any arbitrary object of type "index_type". The definition of such identifiers is redundant and can be deduced from the axiomatic descriptions which follow, but the language designer is forced, by ATLANTIS, to formally introduce the metavariables through such a declaration in order to provide ATLANTIS with sufficient redundant information to ensure suitable and adequate error detection and recovery. It also serves as an aid to a reader of the language definition.

The second subsection of the semantics section of the ADT definition is introduced by the keyword "**axioms**". This keyword is followed by an axiomatic definition of the operations specified by the ADT. The axioms must be consecutively numbered commencing with one. Once again, this is a redundancy mechanism to help ensure that no axioms are inadvertently missed. ATLANTIS checks that the use of each operation is consistent with the syntax given earlier by checking the types, positions and number of arguments. Operations used from other ADT's must be specified in the **where** clause of the ADT definition, otherwise they are flagged as errors. This ensures that a reader is fully aware of what operations are assumed to exist from other ADT's or must be provided by any actual type parameter at the time of the instantiation of the parameterized ADT (to be discussed shortly).

The format of the axioms provided by an ATLANTIS specification are very similar to the format of axioms used in the ADT definitions of Chapter 2. The axioms provided are checked for correct usage and consistency with the specification of the operations. ATLANTIS makes no attempt to prove consistency and sufficient-completeness of any ADT used in the language definition. This aspect is left for the language designer to perform manually and such proofs may be included in the language definition as a comment if so desired. As previously stated, proofs of consistency and sufficient-completeness are generally undecidable problems for arbitrary types and, as a result, it would be inappropriate for the ATLANTIS system to attempt to prove that ADT definitions exhibit these properties.

Another reason that ATLANTIS makes no attempt to ensure that the ADT's used are consistent and sufficiently-complete is that it may well be the language designer intended that they are not! Many current natural language definitions indicate that certain aspects of a language implementation are not specified in the language definition and that implementors have complete freedom of choice over how this aspect will be handled, provided certain guidelines are met. Such aspects are often referred to as implementation dependent features of the programming language. ATLANTIS allows this practice to continue by not demanding that all ADT definitions be consistent and sufficiently-complete. It then becomes the language designer's responsibility to clearly identify any such aspects. It is these aspects of any language definition that result in portability problems with programs written in such a language. Forcing the language designer to highlight such aspects helps to highlight those areas which represent potential portability problems.

ATLANTIS provides six predefined ADT's which can be employed in a language definition. These predefined ADT's are discussed in Section 4.9 and are elaborated in Appendix A. A typical language definition requires a small number of additional ADT's to be defined by the language designer. Appendix C gives details of the ADT's which were required to define the Neptune language; this language definition required the introduction of ten ADT's, of which four were polymorphic. A total of 503 axioms were required to define the 93 operations provided by the ten ADT's.

Once all the necessary ADT's, both polymorphic and monomorphic, have been defined, the polymorphic ADT's can be instantiated to define a new ADT. For example,

ADT integer_table **is new** table [integer, float];

generates a new ADT called "integer_table" which has all of the operations of the polymorphic ADT "table" with the formal type parameter "index_type" replaced by "integer", and "element_type" replaced by "float". Recall from Figure 4.8 that there are certain assumptions regarding the operations provided by the generic formal parameters, namely that the type corresponding to "index_type" must have the operation "equal" defined for it, such that the operation takes two "index_type" arguments and returns a boolean result. The instantiation provided above is substituting "integer" for "index_type" and as a result ATLANTIS checks to ensure that such an operation exists for the integer types. If a suitable operation does exist, then the instantiation may complete, otherwise ATLANTIS raises an error indicating that a suitable operation is not defined for the actual type parameter.

The definition of ADT's and the instantiation of generic ADT's is likely to result in the definition of operators with identical names which may or may not be applicable to objects of differing types returning different results. ATLANTIS uses the context of the usage of an operation to uniquely determine which ADT operation is referred to. Sometimes, however, context will be insufficient to uniquely determine a single candidate operation. In these cases the operation must be qualified by the name of the ADT containing the intended operation. This qualification is performed by naming the ADT in question and the operation from it via a dotted notation, such as:

integer_table.new_table

which specifies the "new_table" operation from the ADT "integer_table" instantiated above. The result is an "integer_table" object, as opposed to an object of any other instance of the table ADT.

4.7 Environment Variables

Once all of the ADT's are defined, the basic information structures for the language definition are in place. It is sometimes convenient to be able to refer to these information structures, or components of them, via some name. ATLANTIS allows the definition of environment variables which permit the language designer to refer to the information structures by name.

Environment variables are introduced by the keyword "**var**", which is followed by an identifier list in which each identifier is separated by commas. The identifier list is followed by a colon, a type name (ADT name) and terminated by a semicolon. The ADT name used may not be the name of a polymorphic ADT, but rather must be a monomorphic ADT (including an instantiation of a generic ADT).

For example, the definitions below:

var int_table_1: integer_table;
var int_table_2, int_table_3: integer_table;

introduce the names "int_table_1", "int_table_2" and "int_table_3" to refer to integer_table objects.

Environment variables can be passed to ADT operations for manipulation and may be assigned the result of an ADT manipulation. Note, however, that ATLANTIS uses the pointer semantics for ":=" operation. Hence, if "int_table_1" refers to one table and "int_table_2" refers to another, as illustrated in Figure 4.9(a), then the assignment statement:

int_table_2 := int_table_1;

changes the situation to that illustrated by Figure 4.9(b).

This indicates that any subsequent change to int_table_1 will have an effect on int_table_2 also. If this is undesirable and it is intended that int_table_2 should refer

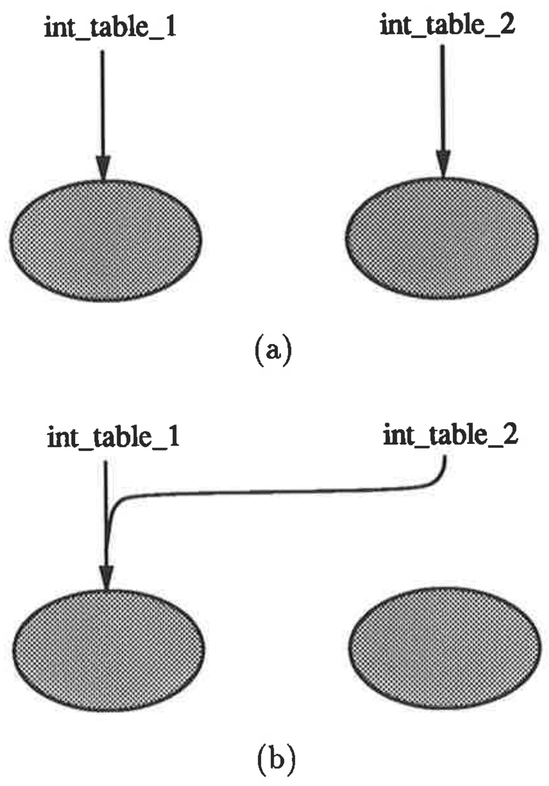


Figure 4.9. Assignment of ATLANTIS environment variables.

to an exact copy of int_table_1 instead then the **copy** operator should be used, for example:

int_table_2 := **copy**(int_table_1);

which replicates int_table_1 and associates this new object with int_table_2. ATLANTIS assumes that the copy operation exists for all ADT's and that it behaves in the prescribed manner. The copy operation and associated semantics need not be elaborated in the ADT definition, but must be present in the ADT implementation. The use of "**copy**" and ":=" makes it clear when the object is duplicated or whether the object referred to by an environment is simply changed to coincide with the object associated with some other environment variable. As a result, potential ambiguity and unexpected results are avoided.

4.8 High Level Operations

High Level Operations (the middle layer of the model) are defined last in ATLANTIS. ATLANTIS allows two forms of HLO – namely procedures and functions. Procedures produce no result but, rather, cause some side-effect on the environment. Procedures are invoked using a **call** statement to distinguish them from function calls, which do return a value and must appear as part of an expression (for example, the right hand side of an assignment statement). Procedures and functions both have access to all of the environment variables declared earlier, and also have access to any variable defined local to the HLO. Both procedures and functions allow parameters to be passed on invocation, but the parameters may not be modified by the HLO, as they are transmitted with the same semantics as **in** parameters in Ada.

As an example of a procedure and a function HLO, the existence of the definition of an ADT known as "ident_stack", which represents a stack of string objects, will be assumed. Further, it will also be assumed, for the sake of this example, that this ADT has the following operations specified within its signature:

new_stack:		$\rightarrow$	ident_stack;
push_stack:	ident_stack $\times$ string	$\rightarrow$	ident_stack;
pop_stack:	ident_stack	$\rightarrow$	ident_stack $\bigcup$ {error} ;
top_stack:	ident_stack	$\rightarrow$	string $\bigcup$ {error};
empty_stack:	ident_stack	$\rightarrow$	boolean;
size_of_stack:	ident_stack	$\rightarrow$	integer;

An environment variable, "id_stk", can then be declared as:

var id_stk: ident_stack;

It is now possible to proceed with the definition of a procedure to reverse the contents of the stack referred to by the environment variable "id_stk" as shown in Figure 4.10. The procedure of Figure 4.10 can be invoked within another HLO, or within a semantic

```
procedure reverse_ident_stack_1 is
   -- Reverse the contents of "id_stk"
   var temp_id_stk: ident_stack;
   begin -- reverse_ident_stack_1
      temp_id_stk := new_stack;
      while not_boolean(empty_stack(id_stk))
      loop
         temp_id_stk := push_stack(temp_id_stk,
                                   top_stack(id_stk));
         id := pop_stack(id_stk);
      pool;
      id_stk := temp_id_stk;
   end; -- reverse_ident_stack_1
```

Figure 4.10. A procedure high level operation.

region embedded in the syntactic definition of the language, by a **call** statement such as:

call reverse_ident_stack_1;

which reverses the contents of the stack as a side effect.

An alternative is to make the HLO a function as shown in Figure 4.11. In this case, it is invoked and the value returned by the function invocation must be used (i.e., it cannot be discarded). The value returned by the function is the contents of the stack referred to by "id_stk", but in reverse order. An example of its use is provided by the assignment below:

id_stk := reverse_ident_stack_2(id_stk);

The **return** clause following an optional parameter list in the function heading indicates the result type of the function. A **return** statement must be executed from within the function body, which consists of the "**return**" keyword followed by an expression which must be of the same type as that specified in the function heading. It is this value that is returned as the value of the function call. The value is returned

```
function reverse_ident_stack_2(stk: ident_stack) return ident_stack is
   -- reverse the ident_stack object and return the result.
   var ident_stack, temp: ident_stack;
   begin -- reverse_ident_stack_2
      temp := copy(stk);
      result := new_stack;
      while not_boolean(empty_stack(temp))
      loop
         result := push_stack(result, top_stack(temp));
         temp := pop_stack(temp);
      pool;
      return result;
   end; -- reverse_ident_stack_2;
```

Figure 4.11. A function high level operation.

at the point that the **return** statement is executed. Any statement occurring after the **return** statement is unreachable and is ignored.

A procedure may also have a **return** statement. In this case, the keyword "**return**" is not followed by an expression and the semantics of the statement is simply the immediate termination of the procedure which executes the statement.

A variety of constructs are permitted within procedures and functions, as well as within the semantic regions interspersed throughout the syntactic definition. These constructs include sequencing via the ";" construct, and assignment as defined by the ":=" construct, as well as conditional and repetitive constructs.

The semantics of the conditional construct provided by ATLANTIS is identical to the conditional construct described in Section 3.2. Its syntactic description is given in Figure 4.12, which shows that the conditional action is introduced by the keyword "**if**" which is then followed by a boolean expression. If the boolean expression evaluates to true, then the action sequence following the keyword "**then**" is executed. If the boolean expression evaluates to false, then the action sequence following the "**else**" is executed. If no such **else** clause exists, then the **if** action as a whole has no effect,

if $\ll$ boolean condition $\gg$
then $\ll S_1 \gg$
[**else** $\ll S_2 \gg$]
fi ;

Figure 4.12. The **if** action within ATLANTIS.

except for any side effects which may have occurred as a result of the evaluation of the boolean condition.

ATLANTIS allows three forms of repetitive construct, again modelled on the constructs discussed in Section 3.2, and hence formal definitions of their semantics are available. The first of these repetitive actions is the infinite loop; this has the syntactic structure shown below:

loop $\ll S_1 \gg$ **pool** ;

In this construct, the actions between the keywords "**loop**" and "**pool**" are executed endlessly. This construct is often used in conjunction with the **exit** construct whose structure is shown below:

exit [**when** $\ll$ boolean expression $\gg$] ;

The **exit** construct is optionally followed by the keyword "**when**", which is terminated by a boolean expression. If the boolean expression evaluates to true, then the loop immediately surrounding the **exit** construct is terminated and execution continues from the action immediately following the keyword "**pool**" which denotes the end of the appropriate loop. If the boolean expression evaluates to false, the entire **exit** construct has no effect. In the case where the **when** clause is not present, the loop surrounding the **exit** construct is unconditionally terminated. The semantics of the **exit** construct only makes sense when the **exit** construct is located within a **loop**–**pool** construct. Any **exit** construct outside such a construct will result in an error as

```
loop
    $\ll S_1 \gg$
    exit when $\ll$ boolean condition $\gg$ ;
    $\ll S_2 \gg$
pool;

    $\equiv \ll S_1 \gg$
      if not $\ll$ boolean condition $\gg$ then
        $\ll S_2 \gg$
        loop
          $\ll S_1 \gg$
          exit when $\ll$ boolean condition $\gg$ ;
          $\ll S_2 \gg$
        pool ;
      fi ;
```

Figure 4.13. Semantics of the **exit** action from within a loop.

```
while $\ll$ boolean condition $\gg$
loop
  $\ll S_1 \gg$
pool;
```

Figure 4.14. Structure of the **while** loop.

the new programming language definition is processed. The **exit** construct may not reside within a HLO invoked from within the **loop–pool** construct. The semantics of an **exit** construct residing within a endless loop is defined in Figure 4.13. The **exit** construct can be used in conjunction with the remaining forms of repetition, which will be discussed now.

The alternative forms of the repetitive construct are the **while loop** and the **for loop**, whose forms are shown in Figures 4.14 and 4.15 respectively. Their semantics are exactly as that discussed in Section 3.2

Whereas the **while** loop of Figure 4.14 requires no further explanation, the **for** loop in Figure 4.15 requires some explanatory remarks. The variable used in the **for** loop is automatically declared, and its value is discarded at the termination of the

```
for ≪variable≫ in ≪lower bound≫ : ≪upper bound≫
loop
    ≪S₁≫
pool;
```

Figure 4.15. Structure of the **for** loop.

```
procedure dump_ident_stack(stk: ident_stack) is
  -- Display the contents of "stk".
  var temp: ident_stack;
  begin -- dump_ident_stack
    -- Take a copy and print the contents of this copy.
    temp := copy(stk);
    while not_boolean(empty_stack(temp))
    loop
      call write_string(top_stack(temp));
      temp := pop_stack(temp);
    pool;
    -- "stk" remains unchanged.
  end; -- dump_ident_stack;
```

Figure 4.16. Display the contents of an identifier stack.

loop. This prevents the language designer from referencing the value of the **for** loop control variable outside the loop. Assignment to, or any other form of manipulation of, the **for** loop control variable is not permitted. As discussed in Section 3.2, if the value of ≪upper bound≫ is less than the value of ≪lower bound≫ then the **for** loop has no effect. ATLANTIS places a further restriction on the **for** loop control variable in that it must be an integer.

These constructs are sufficient to allow the development of useful HLO's within the language definition. Hence, a HLO to display the contents of the "ident_stack" object introduced earlier could be written as shown in Figure 4.16.

The example of Figure 4.16 also highlights one further aspect of ATLANTIS, namely the predefined operations and ADT's provided to the language designer. Figure 4.16 contains calls to the operations "not_boolean" (also used previously in Figures 4.10 and 4.11) and "write_string", both of which are predefined in every ATLANTIS programming language definition. The operation "not_boolean" is a function which takes a boolean argument and returns a boolean value such that the result is the logical negative of the argument. The operation "write_string" is a procedure, and hence invoked with a **call** statement; it takes a single string argument and returns no result. The operation does have the useful side-effect of displaying the argument on the output device (typically the terminal screen). Such primitive operations are necessary when describing the semantics of a "write" statement in languages such as Pascal.

ATLANTIS provides the language designer with a collection of predefined ADT's to aid in language design. These predefined ADT's are discussed in the next section.

Appendix D provides a description of the HLO's which were required in the definition of the Neptune language. Neptune requires 59 HLO's occupying some 2961 lines of the complete Neptune definition of 4547 lines. Clearly, the bulk of the programming language language semantics are contained within the HLO's, shielding most readers from the details of the underlying ADT's.

4.9 Predefined Abstract Data Types

ATLANTIS provides several basic ADT's, and operations on them, in an effort to assist language design and reduce the amount of effort needed to define a language. As the presence of the predefined ADT's is common to all language definitions, the amount of work required by a reader to grasp the essential concepts of a new language is also reduced. Appendix A contains a complete description of the predefined types and the operations made available by them. A natural language narrative is associated with each operation, to provide a description of the operation's behaviour. Some of these operations, such as "return_label", are discussed in detail in the next section as they relate to the run-time behaviour of the ATLANTIS system.

4.10 Implementation of ATLANTIS

As already described, the language definition provided to ATLANTIS is an operational semantic model potentially consisting of several passes. An interpreter based on an ATLANTIS definition need simply build the information structure and manipulate it according to the definition. In this way, the behaviour of the interpreter will precisely mimic the behaviour specified by the underlying information structure model.

Since ATLANTIS specifies where the semantic regions are to have an effect in relation to the syntactic structure of the source program in the newly defined language, all that ATLANTIS need do is produce a parse tree corresponding to the source file, with the nodes decorated with semantic calls which can be invoked at the appropriate time as the parse tree is traversed.

An outline of the ATLANTIS system is shown in Figure 4.17; note that the components which have double outlines are supplied and are used unaltered, whereas those with a single outline are generated from the language definition. The lexical and syntactic parts of an ATLANTIS language definition are carefully checked for internal consistency and translated into input suitable for lex [79] and yacc [66], respectively, in order to generate the appropriate annotated parse tree. In the case of the syntactic component of an ATLANTIS definition, this involves the translation of the EBNF productions to BNF and the merging of rules which share common syntactic derivations, whilst ensuring that the appropriate semantic action can still be located when the interpreter driver traverses the generated parse tree. The generated scanner produces a token stream from a program written in the specified language; these tokens are passed to the generated parser. This parser, in turn, checks the source for syntactic errors and flags them accordingly. If no syntax errors are detected, then a parse tree is produced. The parse tree is annotated with calls to semantic routines produced from the description of the semantics of the language in the input to ATLANTIS. Semantic analysis uses the parse tree, interpreting the program by performing a post-order traversal. Several tree traversals are performed, corresponding to the passes in the semantic definition of the language, with the appropriate semantic actions executed as necessary to build and manipulate the appropriate information structures. The result of the source program is reflected in the state of the information structure when all the necessary passes over the parse tree are complete.

The semantic routines referred to in Figure 4.17 are a pool of Ada routines generated from the semantic actions and HLO's of the language definition. As the underlying model of ATLANTIS is based on ADT's, it is necessary to translate the ADT specifications into Ada packages. This is the only aspect of the language definition for which ATLANTIS does not automatically generate the corresponding implementation component. These ADT's must be manually translated into Ada packages which behave in precisely the manner dictated by the ADT specifications. Although this manual translation is initially labour intensive, this aspect of the interpretive implementation has a high degree of reusability, as the ADT's used to describe one language are often used again in the definition of another language. Hence,

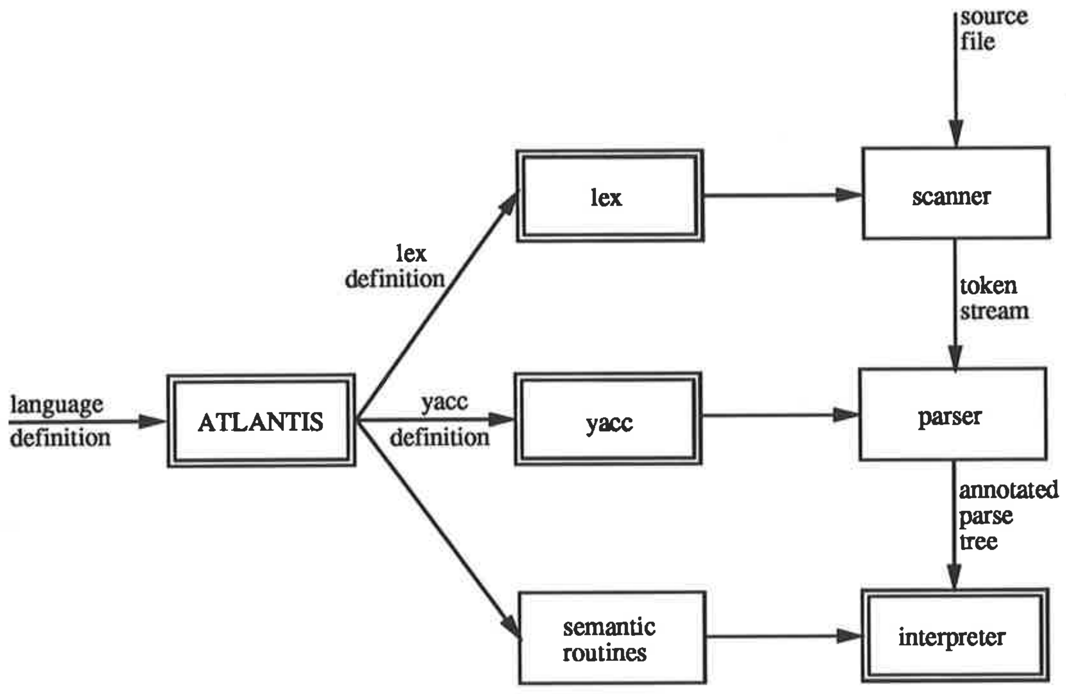


Figure 4.17. Overall structure of ATLANTIS.

an investment of time in the translation of ADT specifications to Ada packages is rewarded later when a subsequent language definition is produced. Commonality of ADT specifications between differing language definitions not only ensures a high degree of reusability, but also results in a firm basis on which different languages may be compared. ATLANTIS may be modified to incorporate techniques for the automatic generation of ADT implementations from ADT specifications using the technology developed for the Larch [54, 154] and OBJ [74, 128] systems.

For each ADT definition in ATLANTIS, there exists an Ada package. For example, the package associated with the table ADT of Figure 4.8 is provided in Appendix B. Such a package must be generic in the case of a polymorphic ADT definition. ATLANTIS presumes that the signature of the operations provided by the package accurately reflects the signature of the ADT definition, that is to say that the operations provided by the package have the same names as those specified in the ADT definition and that the types and ordering of parameters is also preserved, and that the behaviour of the operations corresponds to the axioms. It is also assumed that each package provides a function called "copy", which generates an exact replica of any instance of that type.

Each instantiation of a polymorphic ADT automatically generates the appropriate Ada code. As previously mentioned, ATLANTIS performs the necessary checking to ensure that the instantiation is valid and that the actual type parameters passed provide any necessary operations specified by the polymorphic ADT's definition.

The HLO's and semantic regions of the ATLANTIS definition are automatically translated into Ada functions and procedures. This allows the interpreter driver to invoke them and have them behave in exactly the manner prescribed by the language definition. The necessary annotations to the parse tree are simply the names of the semantic routines as given in the language definition.

The production of a complete interpreter for any newly specified language is complete when the annotated parse tree and a pool of semantic routines is made available to the interpreter driver. This driver is identical for each language and is provided as part of the ATLANTIS system; the role of the driver is to perform as many passes over the parse tree as specified in the language definition, executing the necessary semantic actions where appropriate for each pass. Semantic actions may be attached to leaf nodes representing tokens, or to non-leaf nodes representing a syntactic rule. In each case, the semantic action is executed as a post-order traversal of the parse tree is performed.

As an example, given the rules in Figure 4.18, and "ACC" as input, then the parse tree as illustrated in Figure 4.19 is produced. Performing a post-order traversal over the annotated parse tree results in the execution of the semantic actions at locations corresponding to their place in the rules of the language definition (i.e., Figure 4.18). If the language definition requires multiple passes to correctly specify its semantics, then multiple passes over the parse tree will be required. Figure 4.19 shows the semantic actions, s_i, $1 \leq i \leq 7$, attached to nodes of the parse tree. These nodes may represent terminal or nonterminal symbols of the language. As the tree traversal is performed, the semantic actions are executed as each node is left for the final time (i.e., after the subtrees stemming from the node have been processed). Empty leaf nodes, ϵ, are introduced as needed to provide nodes to which semantic actions may be attached.

```
START:  %% call s1; %%
        R1
        %% call s2; %%
        R2
        %% call s3; %%
        ;
R1      %% call s4; %%
        [ 'A' %% call s5; %% ]
        | 'B' %% call s6; %%
        ;
R2      { 'C' } %% call s7; %%
        ;
```

Figure 4.18. A simple language definition.

If the language definition requires multiple passes to correctly specify its semantics, then multiple passes over the parse tree will be performed. This is the case in the more realistic example presented in Figure 4.20. Figure 4.20(a) gives some syntactic rules

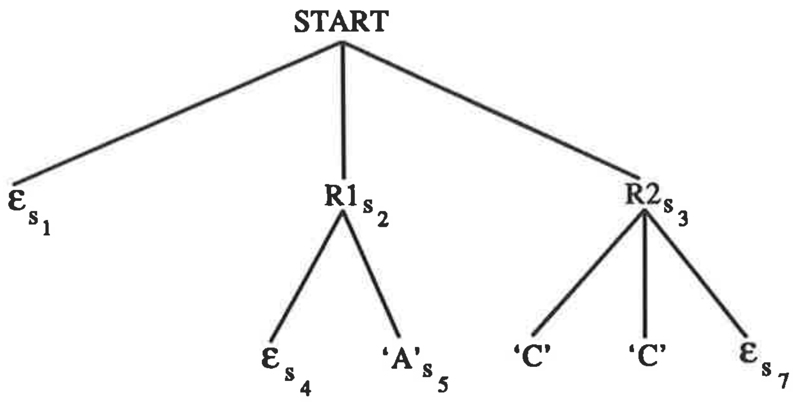


Figure 4.19. The parse tree for the input "ACC".

combined with calls to the semantic routines spread over two passes; Figure 4.20(b) outlines the parse tree that is associated with input such as:

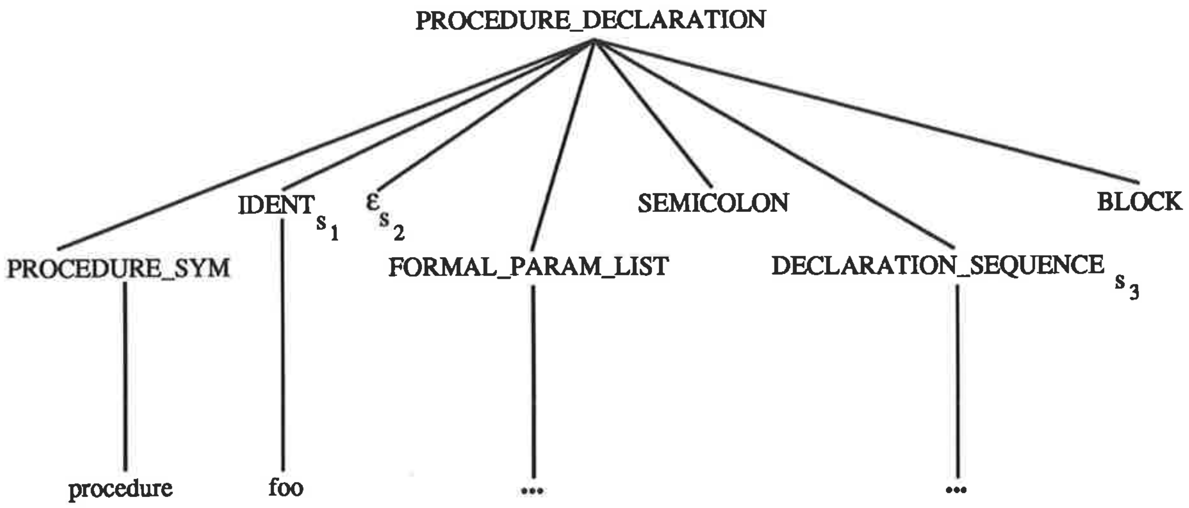
```
procedure foo (a: integer);
  var i: integer;
 begin
   ...
 end;
```

The first pass over the parse tree invokes the semantic routines specified for execution during this pass. This builds an information structure which can be further modified by the semantic routines specified for execution during the second pass over the parse tree. In order to simplify the diagram in Figure 4.20(b), the semantic actions are represented as "s_i". The semantic actions are only executed during the pass in which they are specified as having an effect. If no particular pass is specified, then they are assumed to be effective during the first pass only.

By using the information structure model of the programming language as the basis of the execution of a source program, it is clear that although the interpreter may be inefficient in its execution, it is accurate in its interpretation of the language semantics as it precisely mimics the language definition.

```
PROCEDURE_DECLARATION:
    procedure_sym ident
      %% Pass 1: -- semantic region s1
          call add_procedure_to_symbol_table;
          call create_block;
      %%
      %% Pass 2: -- semantic region s2
          call enter_block;
          call inherit_via_scope_rules(nonlocal);
      %%
      FORMAL_PARAM_LIST semicolon
      DECLARATION_SEQUENCE
      %% Pass 1: -- semantic region s3
          call inherit_via_scope_rules(local);
      %%
      BLOCK;
```

(a)



(b)

Figure 4.20. The rules and the corresponding tree.

In order to describe the dynamic semantics of the language, the language designer must be aware of the underlying mechanism of ATLANTIS; a reader of such a definition must also understand this mechanism. In particular, ATLANTIS must have the ability to continue execution of a program in the defined language from an arbitrary location in the annotated parse tree. This is essential for the description, and execution, of language features such as conditional statements, loops and subroutine calls, where certain statements may need to be omitted or repeated. To meet this need, ATLANTIS provides a special collection of HLO's. To understand how this works, the reader simply needs to be aware that a label may be associated with each node (terminal or nonterminal) or area between two adjacent semantic regions. A label, which is simply a string, can be associated with any node by a call to the *define_label* operation; this operation takes a single argument which is a string and associates it with the current node. The value associated with a node is returned by the function *return_label*; if the current node does not have a label defined, then the previous node which does have an associated label is returned. ATLANTIS also provides a function *generate_name* which returns a unique string which can be used to specify a label name. Allocation of labels to nodes is a dynamic process and is performed as a part of the interpretation of a source program in the subject language. As a result, labels are not built into the parse tree at the time of its generation.

All that remains now is the provision of a mechanism to alter the focus of attention for the interpreter (the locus of control) to some other part of the parse tree, rather than simply progressing to the next node in the tree traversal. Such a mechanism is necessary for the definition of goto statements, conditionals and repetitive constructs. This movement across the parse tree is achieved through the use of the predefined HLO's *goto_next* and *goto_prev*. These operations each take a string as an argument and change the locus of control to the node whose label matches the string specified. The difference between these two HLO's is the direction in which the search is performed – goto_next searches from the current location to the "right" (toward the end of the source program), whilst goto_prev searches in the reverse direction.

It is also necessary to note the label which may potentially be a target of a goto_next or goto_prev HLO. The name of the target label may vary for a number of reasons. For example, a nested **if** statement may have several **else** branches and it is necessary to ensure that the correct one is targetted. The name of the goal label is associated with a node by storing it in an auxiliary label associated with the same node. ATLANTIS provides the operations *define_aux_label* and *return_aux_label* which are analogous to define_label and return_label, respectively. Auxiliary labels simply record the name of a target label which may be the target of a goto_next or goto_prev instruction. These instructions simply locate a node whose label (not auxiliary label) matches the string specified.

4.11 Observations

ATLANTIS is a useful tool for language designers, motivating the production of a formal definition by the generation of an interpretive implementation for the new language. This allows language designers to test language design ideas and ensure they perform in the desired manner. The interpreter simply builds and manipulates the information structures as specified in the language definition; if the observed results of the interpreter are not as expected, then this is a clear indication that there exists some problem with the language definition. Through the use of the language interpreter, the language designer is able to locate and modify those areas of the language definition that are poorly or inadequately defined.

The generation of the interpretive implementation directly from the language definition allows the language designer to experiment with new concepts and features, and also allows the investigation of these features and their interaction with other aspects of the language. This encourages the development of a more refined language definition than might otherwise be obtained through more traditional methods.

Formal methods (such as VDM, denotational semantics and the technique presented in this thesis) share a common benefit to the language designer – if a language feature is difficult to describe, then it is likely to be difficult to learn how to use and is likely to be a source of difficulty in the use of the language. An example of this was shown in Chapter 3: the complexity of describing scope rules in Pascal due to the design decision of handling function value return through an assignment to the function-pseudo-variable, rather than through something like a **return** statement.

Experiments in using ATLANTIS have included the definition of the language Neptune outlined in Section 4.1. As stated earlier, Neptune is essentially Pascal without the following features: labels, constants, type definitions, arrays, records, the character type, variable parameters, with statements, repeat loops, goto statements and trigonometric functions such as sine and cosine. Neptune features not in Pascal are: strings, general loop statement, an exit statement and a return statement. Comments in Neptune take the form of comments in Ada, rather than the style of comments used in Pascal. Despite the aspects of Pascal not included in Neptune, Neptune is still a non-trivial language which contains many of the features of Pascal which are difficult to model accurately (e.g., its scope rules).

The complete Neptune definition requires 4547 lines of which 78 lines describe the lexical components of the language, 169 lines describe the syntactic nature of Neptune, 942 lines define the necessary ADT's, 2961 lines describe the HLO's, and 397 lines associate semantics with syntactic elements.

When processed by the ATLANTIS system, the Neptune definition generates a 236-line file to act as input to lex to produce a scanner; it also generates a 3526-line file to act as input to yacc to produce a parser which will create the decorated parse tree. Also generated are 4672 lines of Ada code, which is then combined with the 9260 lines of Ada common to all interpreters generated by the ATLANTIS system. When compiled and integrated, this results in an interpreter for Neptune which successfully detects and handles syntax errors and which, when presented with a semantically correct Neptune program, produces results which are consistent with the language definition.

The ATLANTIS system itself is relatively simple and easy to use. The generated interpreter is reasonably efficient and it is also helpful with regard to reporting errors in syntactically incorrect programs. The interpreter is faithful to the language definition, with regard to the semantics of the language, detecting and reporting semantic errors only if the language definition indicates this to be the appropriate course of action. When used by the language designer, the generated interpreter is helpful in pinpointing problems in the language definition. These problems include insufficiently defined semantics, and semantics which do not reflect those which were intended by the language designer. The language designer has sufficient feedback to observe a program's execution and deduce, with a great deal of accuracy, where the difficulty within the language definition is to be found. Users who are unfamiliar with the intricacies of the language definition technique find that the generated interpreter behaves in a manner similar to that of other interpreters which which they may have some experience. If a problem exists in the language definition (and hence in the generated interpreter), then a user who is unfamiliar with the language definition is likely to conclude that their Neptune program is in error. Increased support for the language designer in evaluating the language being defined, and refining the the semantics until they match those that the language designer intends, would be useful extensions to the ATLANTIS system and will be discussed briefly in Chapter 6.

The Neptune definition itself is fairly lengthy, yet well structured since it is divided into sections concerned with various aspects of the language definition. However, the definition is still considerably shorter than an accurate natural language description of the equivalent portions of Pascal. The multipass nature of the definition was useful in the definition of various aspects of Neptune and made this task easier, but possibly resulted in a language description which is more difficult to read than may otherwise have been obtained. This is because a reader of the language definition has to understand the state defined by various information structures at the start and end of each pass. This difficulty is overcome to some extent by providing such details as part of the natural language narrative which accompanies the formal description.

The multiple layers of the definition are useful to readers allowing them to moderate the degree of formalism to which they expose themselves, but it is a slight disadvantage from the point of view of the incremental design of a programming language. The ATLANTIS system cannot generate an interpreter for a language until sufficient amounts of each layer are presented. This means that the incremental design of a programming language involves the incremental design and development of each layer of the model simultaneously. There is no mechanism by which all of the necessary ADT's, for instance, can be defined and tested before the HLO's are defined. This may result in the ATLANTIS system being slightly awkward for language designers. ATLANTIS does offer language designers better long-term support than a natural language definition of the programming language, in that the ramifications of minor changes made to the semantics of an almost complete language definition are easily identified. In summary, the language designer faces a greater cost (in terms of effort) to start an ATLANTIS definition of the programming language, in relation to a natural language definition. However, as the language definition nears completion, the ATLANTIS definition of the programming language is significantly easier to manage than a corresponding natural language definition.

In general, the results of the experiment of writing the Neptune definition are encouraging. The ATLANTIS approach results in a language definition from which an interpreter can be generated and, furthermore, the generated interpreter is useful and reasonably efficient. The multi-pass nature of the definition simplifies the language designer's task and, if combined with adequate natural language narrative explaining the interface between the various passes, is relatively simple for readers to comprehend. The multi-layer nature of the model may be awkward for the language designer in the initial stages of the programming language definition, but the ATLANTIS system soon proves extremely useful in assessing proposed semantic changes. From a reader's perspective, the multi-layer approach, when combined with a natural language narrative, is a useful technique which allows each reader to view the language definition with the degree of formalism appropriate to their needs. This makes a programming language defined using the ATLANTIS system useful to compiler writers, language designers and programmers alike.

4.12 Difficulties

ATLANTIS is based on ADT's, which are inherently sequential in nature; as such, they are of little use in describing parallel language features. The difficulty is that the integrity of the data objects specified by the ADT cannot be guaranteed in an environment where multiple processes may attempt to access the data object at any one time. Another technique, known as shared data abstractions (SDA's), rectifies this problem and is explored in the next chapter. Through the use of this technique, it is then possible to formally describe the semantics of parallel programming languages with a layered semantic model of the kind already described in this thesis. Such a model is also illustrated in the next chapter.

Chapter 5

Describing Parallel Languages

5.1 Introduction

The model presented in the previous two chapters is founded on regarding the information structure involved in an operational semantic model as an ADT; this ADT and its associated primitive operations are then specified algebraically, thus significantly increasing the precision of the operational semantic model.

Algebraic specification of ADT's traditionally assumes a sequential environment for the application of the operations of the ADT. However, the information structures required for the description of a parallel programming language must have the ability to be accessed by multiple concurrent processes. As a result, ADT's no longer provide a wholly suitable medium for the basis of a model for the description of parallel programming languages.

There is an abundance of methods for describing the semantics of parallel programming languages (for example, automata [102], modal logic [1], temporal logic [56, 117], attribute grammars [140], and axiomatic semantics [5, 108, 109, 110, 126]). However, the various formal descriptions of parallel programming languages tend to be rather difficult to read and difficult to write.

Adoption of any of the techniques described above would result in discarding the model developed for sequential languages. Fortunately, a technique has been developed by Mallgren [80] for the specification of shared data abstractions (SDA's), which permit the description of ADT's which operate in an environment where multiple processes may attempt to access the data structure simultaneously.

5.2 Mallgren's Approach to Shared Data Abstractions

The extension of sequential abstract data types to a parallel programming environment requires the addition of a mechanism to synchronize (possibly conflicting) accesses to the shared object by several independent parallel processes. Shared data abstractions offer a mechanism to specify this synchronization within an algebraic framework. The use of Mallgren's technique for SDA's will now be introduced, using as an example the abstract data type specification for an infinite queue (shown in Figure 5.1). For the purposes of this illustration, it will be assumed that the queue data structure is being shared by some number of parallel processes; the elements which can be stored in the queue are of type "elem". A proof of correctness of the SDA in Figure 5.1 will also be given.

Each SDA object consists of a protected resource, called "state", with one or more ADT operations (which Mallgren calls "events" and which will be called *basic state generators* following Freidel [33]) which may be applied to the protected resource. These basic state generators are applied to the state at the request of an auxiliary

Infinite_Queue_SDA (elements of type "elem")

high level routines

get		$\rightarrow$	elm
put	elem	$\rightarrow$	

basic state generators

get\$c	state	$\rightarrow$	state
get\$r	state	$\rightarrow$	state
put\$c	elem $\times$ state	$\rightarrow$	state
put\$r	elem $\times$ state	$\rightarrow$	state
\$init		$\rightarrow$	state

auxiliary functions

queue_size	state	$\rightarrow$	integer
fetch_item	integer $\times$ state	$\rightarrow$	elem $\bigcup$ {error}

axioms

for all N **in** integer, L **in** state, X **in** elem, **let**

```
get
     :wait(L)   =  (queue_size(L) = 0)
     :value(L)  =  fetch_item(queue_size(L), L)
put
     :wait(X, L)  =  false

(1)   queue_size($init) = 0
(2)   queue_size(get$c(L)) = queue_size(L)
(3)   queue_size(get$r(L)) = queue_size(L) - 1
(4)   queue_size(put$c(X, L)) = queue_size(L) + 1
(5)   queue_size(put$r(X, L)) = queue_size(L)
(6)   fetch_item(N, $init) = error
(7)   fetch_item(N, put$c(X, L)) = if N = 1
                                    then X
                                    else fetch_item(N-1, L)
                                    end if
(8)   fetch_item(N, put$r(X, L)) = fetch_item(N, L)
(9)   fetch_item(N, get$c(X, L)) = fetch_item(N, L)
(10)  fetch_item(N, get$r(X, L)) = fetch_item(N, L)
```

Figure 5.1. The shared data abstraction specification of an infinite queue.

function being applied to the state, or due to a ***high level routine*** (called an "operation" by Mallgren) being executed on the shared object. These high level routines are the means by which external processes access the shared data object.

Each high level routine implicitly applies two basic state generators to the "state" object, as a result of the invocation of a high level routine. These particular basic state generators will be called *event functions* (or simply *events*). The first, a *call* event, is applied immediately to the state when the high level routine is invoked. The second, a *return* event, is applied immediately before the high level routine returns to the calling process. A process executing a high level routine on an SDA object may be forced to wait if the *return* event cannot be evaluated immediately. There are two high level routines in the SDA in Figure 5.1, namely "get" and "put"; for each of these, there is a *call* event and a *return* event, as summarized in Table 5.1. Thus, for example, when an queue object with internal state "S" is accessed with the SDA routine "get", the state becomes "get$c(S)". When the "get" SDA routine returns, assuming that the internal state is "S′" immediately before returning, the result is "get$r(S′)". The internal state of an queue object is implicitly initialized by "$init" when the object is created.

High level routine	**Call event**	**Return event**
get	get$c	get$r
put	put$c	put$r

Table 5.1. Call and return events for the infinite queue shared data abstraction.

In addition to the usual kinds of axioms which occur in algebraic specifications of ADT's (represented by axioms 1 through 10 in this case), the axioms in the queue description in Figure 5.1 also include the specification of two *characteristic functions* associated with each high level routine (and thus their corresponding events). The first of these characteristic functions, "wait", describes the synchronization and sequencing requirements for the completion and return of the called high level routine. As can be seen in Figure 5.1, the "get" function, when called, may not return while the number of values in the queue (defined by the function "queue_size") is zero; that is, it must wait until there exists at least one value in the queue. On the other hand, the "put" SDA routine need never wait before returning. The second characteristic function is called "value" and describes the value returned by the corresponding SDA routine when it does return from a call. In the case of the "get" SDA routine, the value at the head of the queue is returned. The "put" routine does not return a value and so no "value" characteristic function specification is given for this routine; it simply has the side effect of adding a value to the tail of the queue.

The evaluation of characteristic functions is considered to be atomic. They are not interruptible by another process wishing to evaluate the same function. It is also unnecessary to specify the result of the characteristic function applied to every possible state. This is because, for the sake of brevity and readability, the SDA specification includes implicit axioms applying the characteristic function to the previous state if no rule is applicable for the current state.

Auxiliary functions, like the basic state generators, are invisible outside the SDA. The purpose of the auxiliary functions is to facilitate the specification of the characteristic functions. For example, the auxiliary function "queue_size", defined by axioms 1 to 5 of Figure 5.1, exits so that it can be used in the specification of the "wait" and "value" characteristic functions for the "get" routine. Similarly, axioms 6 to 10 define a "fetch_item" function which is needed in the specification of the "value" function for "get". In fact, the sole purpose of the axioms 1 to 10 is the definition of the semantics of these auxiliary functions.

Thus, the internal state of the queue may be denoted by the sequence of SDA routines called and returned, sequenced in accordance with the synchronization constraints implied by the "wait" characteristic functions for each SDA routine. Auxiliary function applications are not recorded in this representation of the internal state.

Some comments on the domains and ranges of these various functions are in order at this point. The notation used for the basic state generators and auxiliary functions is that normally used in specifying the operations of ADT's; notice, in particular, that the type "state" occurs frequently in the specification of these functions in Figure 5.1. A little more unusual is the notation employed for the specification of the high-level routines; following the convention used in [80], the "state" type is omitted from these operations, in order to simplify later specifications.

For an abstract data type to be shown correct, there are two issues which must be dealt with: sufficient-completeness and consistency. The same properties are required of a shared data abstraction. For the queue SDA, the canonical form of "state" objects contains one "$init" operator and zero or more applications of event functions (i.e., zero or more applications of "get$c", "get$r", "put$c" and "put$r").

Consider the properties of sufficient-completeness and consistency for the transfer operators "queue_size" and "fetch_item". For "queue_size", axioms 1 through 5 are relevant. Axiom 1 defines the initial case and axioms 2 through 5 define the general cases in the induction process (one for each of the operations listed above). Thus, the application of "queue_size" to any object of type "state" results in at least one value. Note also that the application of "queue_size" to any object of type "state" results in at most one value, because there is only one applicable axiom in each case. The axioms are thus sufficiently-complete and consistent with respect to "queue_size". A similar argument, using axiom 6 and axioms 7 through 10, verifies the specification of "fetch_item". Since the basic constructors are not related to each other by any axioms, we can now state that this specification of an infinite queue is sufficiently-complete and consistent.

It is important to point out that Mallgren's notation focuses primarily on the high-level routines. This makes it amenable to use in the description and design of programming languages, as demonstrated in [105, 107], where the desired view of the shared object is that from the perspective of the processes which use the object, without the detail of the sequence of ADT operations and synchronization events which the high level routines may entail.

As seen in the previous chapter, ATLANTIS uses ADT's to define data types and the operations on them. These ADT definitions are then used to define variables which can be manipulated by the language definition to describe the semantics of the programming language; these variables are simply named values which may be passed to an operation and altered in some way before a new value is assigned or associated with the variable name. SDA's differ from this behaviour in that the approach advocated by Mallgren does not export a type name which can be used to declare variables; rather, each SDA definition describes a single object of the defined type and its "state" is managed by the SDA. In essence, the SDA is specifying an object and protecting it by hiding the state of the object.

5.3 The Ada Rendezvous

This chapter will present a description of a detailed, formal model of the intertask communication in Ada as an example of the use and power of SDA's in the description of parallel programming languages. First, however, an examination of the various forms of communication in Ada is warranted. The various forms, also discussed in [33, 86, 87], are based on the various language constructs involved with Ada's rendezvous mechanism. This mechanism is achieved through entry calls and **accept** statements, with certain kinds of nondeterminism being introduced through the use of Ada's **select** statement.

Ada provides three kinds of entry call: deterministic entry calls, conditional entry calls and timed entry calls. Deterministic entry calls are, as the name suggests, deterministic in nature and will eventually occur provided that program execution lasts sufficiently long. Conditional and timed entry calls are similar in behaviour and both occur within **select** statements. In each case, the entry call may or may not eventually take place. In essence, there are then two distinct forms of entry call which are termed, following [33, 86, 87], *deterministic send* and *conditional send*; these cover entry calls made outside and within a **select** statements, respectively.

There are two forms of **accept** statement which occur in Ada: deterministic **accept** statements, which do not reside within **select** statements, and nondeterministic **accept** statements, which do reside within **select** statements. This latter possibility arises through the "selective wait" form of the **select** statement and provides a "nondeterministic" selection from several alternatives which are otherwise ready to proceed. These two types of **accept** statement are termed *deterministic receive* and *nondeterministic receive*, respectively.

In order to minimize the complexity of the present description, the model presented here only concerns itself with **in** and **out** parameters of the rendezvous. The model can be extended to cater for **in out** parameters. Further, the model does not consider exceptional circumstances which may occur, such as a task becoming abnormal. The focus here is on the rendezvous, rather than on the description of various other aspects of the different forms of the **select** statement. Thus, the above classification scheme represents a suitable level of abstraction.

Combination of the various forms of entry call with the various forms of **accept** statement provides four types of rendezvous. The current definition of Ada provided by the Ada language reference manual (ALRM) [141] indicates that *synchronization* of the tasks must occur before any form of *communication* can take place. Any formal model of the Ada rendezvous mechanism must address synchronization, in addition to describing the communication which occurs between the tasks involved.

In the ensuing discussion, reference will be made to various paragraphs of the ALRM[1] which support the description of Ada presented here. The ALRM provides the following statements when defining the rendezvous:

> ALRM 9.5(6): The parameter modes defined for parameters of the formal part of an entry declaration are the same as for a subprogram declaration and have the same meaning (see 6.2). The syntax of an entry call statement is similar to that of a procedure call statement, and the rules for parameter associations are the same as for parameter calls (see 6.4.1 and 6.4.2).
>
> ALRM 9.5(10): Execution of an accept statement starts with the evaluation of the entry index (in the case of an entry of a family). Execution of an entry call statement starts with the evaluation of the entry name; this is followed by any evaluations required for actual parameters in the same manner as for a subprogram call (see 6.4). Further execution of an accept statement and of a corresponding entry call statement are synchronized.

[1]Paragraphs of the ALRM will be referred to using the notation "ALRM $x.y(z)$" where x is the chapter number, y the section number and z the paragraph number; a section of the ALRM will be referred to using the notation "ALRM $x.y$".

ALRM 9.5(14): When an entry has been called and a corresponding accept statement has been reached, the sequence of statements, if any, of the accept statement is executed by the called task (while the calling task remains suspended). This interaction is called a *rendezvous.* Thereafter, the calling task and the task owning the entry continue their execution in parallel.

ALRM 9.5(15): If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued; there is one queue associated with each entry. Each execution of an accept statement removes one call from the queue. The calls are processed in the order of arrival.

An examination of the above paragraphs indicates that there is a queue associated with each entry, such that each entry may only communicate with one task at a time. We know from elsewhere in the language definition that the calling task knows the name of the task it calls, as well as the name of the entry, but that the called task knows nothing about its caller. That is to say, the rendezvous mechanism of Ada is asymmetric.

Paragraph 9.5(10) of the ALRM suggests that the calling task must evaluate the actual parameters before the rendezvous commences. This suggests that the transmission (communication) of the **in** parameters forms part of the rendezvous and hence occurs synchronously. Paragraph 9.5(6) informs us that the rules for parameter association are the same as that for subprogram calls and refers us to ALRM 6.4.1 and ALRM 6.4.2. As the section of the ALRM on tasking provides no special information on the semantics of parameter transmission in the context of a rendezvous (such as when **out** mode parameters are transmitted), the reader is forced to check the sections referred to. There, the ALRM states:

> ALRM 6.4.1(7): • After (normal) completion of the subprogram body: for a parameter of mode **in out** or **out**, it is checked that the value of the formal parameter belongs to the subtype of the actual variable. In the case of a type conversion, the value of the formal parameter is converted back and the check applies to the result of the conversion.
>
> ALRM 6.4.1(10): The exception CONSTRAINT_ERROR is raised at the place of the subprogram call if either of these checks fails.

which suggests, for example, that the value of the **out** parameters are transferred *after* normal completion of the subprogram block. To place this into perspective with respect to a parallel environment demands an explanation of "normal completion" with respect to tasks. An explanation of the latter concept is to be found in:

> ALRM 9.4(5): A task is said to have *completed* its execution when it has finished the execution of the sequence of statements that appears after the reserved word **begin** in the corresponding body. Similarly a block or a subprogram is said to have completed its execution when it has finished the execution of the corresponding sequence of statements. For a block statement, the execution is also said to be completed when it reaches an exit, return, or goto statement transferring control out of the block. For a procedure, the execution is also said to have completed when a corresponding return statement is reached. For a function, the execution is also said to have completed after the evaluation of the result expression of a return statement. Finally, the execution of a task, block statement, or

> subprogram is completed if an exception is raised by the execution of its sequence of statements and there is no corresponding handler, or, if there is one, when it has finished the execution of the corresponding handler.

Unfortunately, this does not help to place the phrase "normal completion of the subprogram body" in ALRM 6.4.1(7) into context with respect to a rendezvous. Hence, our only choice is to infer that the equivalent statement with respect to the rendezvous mechanism would be "normal completion of the rendezvous". Reference to ALRM 9.5(14) suggests that the rendezvous is complete when the accept statement has been executed, at which point the called and calling tasks continue their execution in parallel. This suggests that, as far as the called task is concerned, the transmission of **out** parameters occurs asynchronously. This is further supported by the fact that ALRM 9.4(5) states that an exception raised within a task results in the completion of that task or block. Since ALRM 6.4.1(10) tells us that failure of any of the constraint checks on the **out** parameters results in a CONSTRAINT_ERROR being raised at the point of the subprogram call (substitute entry call), we again find indirect evidence to suggest that the transmission of **out** parameters may occur asynchronously outside of the rendezvous, but before another rendezvous with that entry may commence.

If our interpretation of "normal completion of the subprogram body" of ALRM 6.4.1(7) is "normal completion of the statements inside the accept statement, but before the rendezvous is considered to be finished", then the transmission of **out** parameters is clearly synchronous. This is substantiated by considering ALRM 6.4.1(7) and 6.4.1(10), which dictate that an exception is to be raised at the point of the subprogram call (entry call).

Consideration of ALRM 9.4(5) and 6.4.1(10), which both describe behaviour with respect to exceptions, results in a contradiction. If an exception is raised during the rendezvous, it must affect the called task according to ALRM 9.4(5), whilst ALRM 6.4.1(10) indicates that a constraint check failure during transmission of **out** parameters only results in a CONSTRAINT_ERROR within the calling task. The only conclusion we can draw is that transmission of **out** parameters may be synchronous (part of the rendezvous) as far as the calling task is concerned, but may be asynchronous (outside the rendezvous) with respect to the called task. This seeming contradiction arises as a direct result of attempting to define a parallel language feature by analogy to a sequential language feature, in conjunction with an imprecise definition as to the exact moment that a rendezvous commences and finishes. This leaves a reader to "fill in the gaps", resulting in many compiler writers drawing on their sequential language experiences on monoprocessor architectures and implementing **out** parameters as though they form part of the rendezvous, thus ignoring the contradiction between ALRM 9.4(5) and 6.4.1(10). The mere fact that many compiler writers have chosen this option does not mean that it is correct, nor that it is the only possible interpretation. The complexity of intertask communication is a result of not viewing the communication aspect as a critical element and modelling it in a suitable fashion before the description of tasking in the ALRM was attempted.

5.4 A Shared Data Abstraction Model for the Ada Rendezvous

A formal model of the Ada rendezvous must address the issues raised in the previous section, as well as describing the synchronization occurring during a rendezvous attempt. The model presented in [33], which forms the basis of the discussion here, represents a potential rendezvous between two tasks by an *interface object*, consisting of:

- a synchronous communication port called "info_in" (to cater for **in** parameters), and
- an asynchronous communication port called "info_out" (for **out** parameters).

The model is illustrated in Figures 5.2 and 5.3. Figure 5.2 provides an outline of the definition for three Ada tasks: the task "T1" merely accepts a call on its entry "e1", task "T2" accepts a call on its entry "e2" and calls the entry "e1" belonging to "T1", and "T3" calls entry "e1" of "T1" and entry "e2" of "T2". The interactions between these tasks are represented by the two interface objects on the lefthand side of Figure 5.3. The info_in port handles two essential aspects of the interaction between tasks: the queuing of requests to entry points and the synchronization of message transfer. In spite of the fact that a rendezvous only involves two tasks, it is necessary to model the queuing of tasks on an entry call because Ada allows several calling tasks to simultaneously attempt to rendezvous with a given entry.

Freidel [33] models the tasks themselves as *communication lists*. There is one of these for each task instance and a communication list is divided into two tables:

- the *in_list* table, which describes the entries belonging to the task and indicates the corresponding interface object (of which there will be precisely one for each entry), and
- the *out_list* table, which describes the entries of other tasks potentially called by this task – the elements of this table indicate the appropriate interface objects.

```
Task T1;
   ...;
   accept e1;
   ...;


Task T2;
   ...;
   accept e2;
   ...;
   T1.e1;
   ...;


Task T3;
   ...;
   T1.e1;
   ...;
   T2.e2;
   ...;
```

Figure 5.2. A sample Ada program.

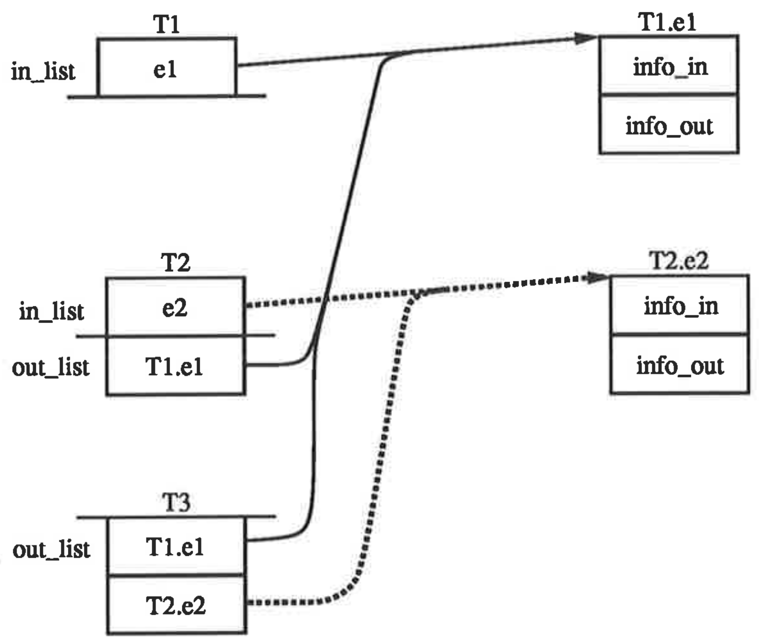


Figure 5.3. Communication paths of Ada tasks.

The communication lists for the tasks in Figure 5.2 are shown on the lefthand side of Figure 5.3. The part of the list above the longer horizontal line is the in_list table, whereas that below the longer line is the out_list table. Either of these tables may, of course, be empty for a particular task, as illustrated by the fact that the out_list for "T1" and the in_list for "T3" are both empty.

Figures 5.2 and 5.3 are sufficient to indicate that several tasks may attempt to access the queue associated with a particular entry at any given time. For example, instances "T2" and "T3" may attempt to access the queues associated with the entry "e1" of "T1" at the same time. Such queues must be protected to ensure data integrity and suitable behaviour. In this context, ordinary abstract data types are insufficient. The model described here, first described by Freidel in [33] and later refined in [86], uses the notion of SDA's discussed in Section 5.2 to represent such objects.

Operations on info_in ports are modelled by the SDA "Synchronous_SDA", which is outlined in Table 5.2; operations for the info_out ports are modelled by "Asynchronous_SDA", outlined in Table 5.3. The "PID" referred to in Tables 5.2 and 5.3 denotes a process identifier type; these process identifiers are used to uniquely identify a task instance in the queue within the SDA. The reader is referred to [34] for a complete specification of these SDA's using Mallgren's approach, and for a proof of their consistency and sufficient-completeness. The proof of consistency and sufficient–completeness is similar to that discussed for ADT's in Chapter 2.

To the user of the SDA, it is the high level routines that are of prime importance. These are basically divided into two groups: those associated with writing to the interface, and those associated with reading information from it. Within each group, it is necessary to consider two cases: the deterministic case and the conditional case.

Operation	**Description**
write_try(PID)	Place the process name specified by *PID* into the queue. This operation informs the shared data abstraction that the indicated process wishes to write to the communication port.
can_write_complete?(PID)	Checks the queue and immediately returns a boolean value indicating if the indicated process could complete a write operation without any waiting.
write_wait(PID)	Checks the queue and waits until the indicated process can validly write to the communication port without waiting.
write(PID, A)	Allows the process named *PID* to write the information *A* to the communication port. The operation is forced to wait if another process is currently writing to the port.
read_try	Informs the communication port that there is a process wishing to read information from it. No attempt is made to get the information.
can_read_complete?	Checks the communication port and returns a boolean value indicating if there is anything on the queue to be read.
read_wait	Forces the calling process to wait until there is information to be read from the communication port. No attempt is made to read it.
read	Reads the current information from the communication port.

Table 5.2. The operations applicable to a synchronous communication port.

Operation	Description
read	If there is a value to read in the communication port, it is read and the value returned; otherwise, the calling process is forced to wait until there is a value to read.
write(A)	Write the value A to the communication port. The writer must wait if the port is actively involved in another reading or writing operation.
can_read?	Immediately returns a boolean value indicating if the caller could successfully complete a read operation without waiting.

Table 5.3. The operations applicable to an asynchronous communication port.

For processes wishing to place information into the interface, it is necessary to first request permission to write into the interface using the "write_try" operation. This queues the requesting process in the interface. The process then has two choices. In the deterministic case, the operation "write_wait" is called. This returns to the task when the operation "write" can be performed successfully – the task must then call "write". For the conditional case, the task can call "can_write_complete?". This operation performs queue management, as well as inquires about the state of the interface. If the "write" operation can complete, the operation returns "true"; if not, it returns "false" and removes the task from the queue. The operations for message reception are similar.

5.4.1 Modelling Interface Objects and Communication Lists

As mentioned above, an interface object represents a potential rendezvous between two tasks. It uses its info_in field to record the path along which information is passed into the called task and its info_out field to record the path along which the **out** parameters are passed. Interface objects are regarded as being variables of the type "interface" defined in Figure 5.4. As shown in this figure, the info_in field is occupied by an instance of the SDA called "Synchronous_SDA" (whose operations were outlined in Table 5.2), and the info_out field is occupied by an instance of the SDA "Asynchronous_SDA" (whose operations were given in Table 5.3).

Figure 5.4 also defines a type describing a communication list. This shows that such a list is composed of an "in_list" table and an "out_list" table. The elements of the in_list table represent the entries of the task and their corresponding interface objects. This table is an object of type "In_Table_ADT", whose operations are summarized in Table 5.4. As shown in Figure 5.4, in_list is a table indexed by a string (representing the entry name) and whose elements are pointers to interface objects. Associated with each entry is a parameter list which is represented by an object of type "Param_Info". Since parameter transmission is not the focus of attention in the description of the inter-task communication of Ada, details of the type "Param_Info" will not be discussed further. The SDA in the "info_in" field of the corresponding interface is used to receive the parameters passed by the call being accepted.

The elements of the out_list table represent entry calls on other tasks. As a call to an entry in another task involves specifying the task involved as well as the entry, the table is indexed by both the task name and the entry name. The related "info_in" field of the corresponding interface is used to send information at the time of the call, while the "info_out" field is used to send the reply information. The out_list table is of type "Out_Table_ADT", whose operations are summarized in Table 5.5.

In order to complete the information structure model of the semantics of intertask communication in Ada, it is now necessary to provide adequate descriptions of the

```
type
  interface =
      record
        info_in: Synchronous_SDA(Param_Info);
        info_out: Asynchronous_SDA(Param_Info);
      end;

  communication_list =
      record
        in_list: In_Table_ADT(string, ↑interface);
        out_list: Out_Table_ADT(string, string, ↑interface);
      end;
```

Figure 5.4. Types used in the description of Ada tasks.

Operation	**Description**
create	Make a new "In_Table_ADT" object.
insert(T, E, I)	Insert into table *T*, an entry with index *E*, and associated information *I*. If such an index already exists in the table, then alter the information associated with it.
delete(T, E)	Delete from the table *T*, the information associated with the index *E*. If *E* does not exist in table *T*, then this operation has no effect.
retrieve(T, E)	Return the information associated with the index *E* in table *T*. If *E* does not exist in table *T*, then raise an error.
is_defined(T, E)	Return "true" if the index *E* is defined for table *T*; otherwise, return "false".
is_empty(T)	Return "true" if the table *T* is empty; "false" otherwise.

Table 5.4. The operations applicable to an "In_Table_ADT" object.

Operation	Description
create	Make a new "Out_Table_ADT" object.
insert(T, P, E, I)	Insert into table *T*, an entry with indices *P* and *E*, and associated information *I*. If such an index already exists in the table, then alter the information associated with it.
delete(T, P, E)	Delete from the table *T*, the information associated with the indices *P* and *E*. If the indices *P* and *E* do not exist together in table *T*, then this operation has no effect.
retrieve(T, P, E)	Return the information associated with the indices *P* and *E* in table *T*. If *P* and *E* do not exist together in table *T*, then raise an error.
is_defined(T, P, E)	Return "true" if the indices *P* and *E* are defined together for table *T*; return "false" otherwise.
is_empty(T)	Return "true" if the table *T* is empty; return "false" otherwise.

Table 5.5. The operations applicable to an "Out_Table_ADT" object.

transformations of the information structures caused by the various relevant language features. This is done by giving an algorithmic description of an *event* corresponding to each such language feature. These events describe the semantics of communication between Ada tasks by setting up interfaces and communication lists, linking them, rearranging their interconnections, and transferring values to and from the interfaces. Each of these algorithms is regarded as a set of actions executed in place of the language feature it describes. Note that extracting a value from an interface may cause the task instance executing the language feature being described to wait if the shared data abstraction in the interface object dictates. In such a case, the evaluation of the corresponding event is suspended at the point of attempting to extract the value. When a value becomes available, the evaluation of the event may continue.

The notation used in the algorithmic descriptions derives largely from that used in [83]. The identifier "current" will always be a pointer to the communication list corresponding to the task instance executing the event for language feature being described. In this paper, it will represent either the calling task or the called task, depending on whether a send or a receive event is being described. Other aspects of the notation used are as follows:

- Each description will be delimited by the keywords "**begin**" and "**end**".
- Sequential actions will be separated by semicolons ";".
- Variables which are local to the events being described are always in single uppercase letters, such as "A".
- The symbol ":=" will be used to denote assignment.
- "Dot notation" is used for selecting fields within this model. Thus, for example, if "B" is an interface object, then "B.info_in" refers to the "info_in" field of the interface object "B". This notation is also used to uniquely reference an operation whose definition may be overloaded due to its provision by several ADT's.
- The notation "$\uparrow$" means that the reference is followed to the object it refers to. Thus, if "A" refers to an interface object "B", then "A$\uparrow$" is the interface object.
- There is a predefined value used in the model. The value "null" indicates that the field is not used; although the value "null" indicates emptiness, it may be dereferenced (also giving the value "null") or have field selection performed on it (again giving the value "null"). There is also a predefined tyoe: the type "string" (already seen in Figure 5.4) indicates a field composed of a character string of arbitrary length.

- The conditional and repetitive constructs used have closing keywords, such as "**if** ... **fi**" . (Keywords will always be in lower case letters and in bold face).
- The notation "−−" will be used to introduce comments, which continue until the end of the line.

5.4.2 Primitives

The algorithmic event descriptions introduced above make use of various primitives in the manipulation of the information structures. These primitives frequently take one or more arguments which can be regarded as being transmitted according to the familiar value transmission mechanism. These primitives could be defined formally using an approach similar to that of [85, 103, 104, 106].

In order to return information from event descriptions, we introduce the primitive:

return(B, X)

The first parameter, "B", is a boolean value that indicates if the rendezvous completed successfully, and the second parameter, "X", is any information that needs to be returned to the task instance invoking the event. Execution of the "return" primitive results in the immediate termination of the event and the values "B" and "X" are made available to the calling event.

The model also employs the primitive object:

STATEMENT

which may be used in any event to retrieve the statement associated with the event. These "statement objects" are composed of four fields. The "task" field indicates (if possible) the task named in the statement. The "entry" field indicates the entry

```
type
   process_identification =
      record
         name: string;
         comm_list: ↑communication_list;
      end;
```

Figure 5.5. Types used in the description of Ada tasks.

name used in the statement. The "in_params" field indicates the **in** parameters for entries called, while "out_params" indicates the **out** parameters. Because we are not concerned with the number or types of parameters in this model, we will use these single fields to denote zero or more parameters.

The variable:

FORMAL_PARAMETERS

allows the events associated with accept statements to access the information transferred on entry call.

The variable:

current

is used to represent the current process which is executing the event in which the variable current is referenced. Each process, for the purposes of this model, will be represented as an object of type "process_identification", shown in Figure 5.5. This figure shows that each process has a name associated with it, as well as an object of type communication_list for the transfer of information to and from the process. It is assumed that a unique name is generated for every task in the executing Ada program.

5.4.3 Communication Events

As discussed in Section 5.3, Ada can be regarded as having two groups of communication events, each group containing two events. First, there is a group concerned with message sending, which can be carried out either deterministically or conditionally. The other group is concerned with message reception, which can be deterministic or nondeterministic. Thus, we provide descriptions for the following events:

(1) Deterministic_Send,

(2) Conditional_Send,

(3) Deterministic_Receive, and

(4) Nondeterministic_Receive.

In terms of language features, events (1) and (2) relate to entry calls, whereas events (3) and (4) arise because of accept statements. Consistent with the approach adopted in Section 5.3, each of the communication events focuses on the communication which takes place during the corresponding kind of rendezvous. No attempt is made to describe other aspects of the semantics of, say, the **select** statement.

For tasks wishing to place information into an interface object (recall Figure 5.4, and Tables 5.2 and 5.3), it is necessary to request permission to write into the interface using the "write_try" operation. This queues the task in the interface. The task then has two possible paths. In the deterministic case, the operation "write_wait" is called. This returns to the task when the operation "write" can be performed successfully; the task must then call "write". For the conditional case, the task can call "can_read_complete?". This operation performs queue management as well as inquiry into the state of the interface. If the "write" operation can complete, the operation returns "true"; if not, it returns "false" as well as removing the task from the queue. The above sequence of calls ensures that the parameter information is passed to the called task after the rendezvous commences and not before. The operations for message reception are similar, but without the queuing of tasks.

There are four different events for communication. "Deterministic_Send" and "Conditional_Send" relate to entry calls; "Deterministic_Receive" and "Nondeterministic_Receive" correspond to **accept** statements. Furthermore, the "Conditional_Send" and "Nondeterministic_Receive" events are tied to the **select** statement. The fact that the "rendezvous" concept allows a two-way transfer of information creates a transaction and is modelled by the use of two SDA objects, one such object for each direction of transfer.

There are essentially two reasons why inter-task communication via the Ada rendezvous is complex. The first is nondeterminism. The other is that, conceptually, information does not "flow" between the tasks – the rendezvous directly transfers information between tasks.

5.4.3.1 The Deterministic_Send Event

Consider the statement:

T1.E1(A, B);

where "T1" indicates some task instance and "E1" indicates an entry within that instance. Note that this entry call is deterministic, in that the calling task will suspend execution until the call is accepted by "T1".

Figure 5.6 shows the algorithmic description of the Deterministic_Send event, which is the event corresponding to this language feature. First, the name of the entry

```
Deterministic_Send =
    begin
      E := STATEMENT.entry;
      T := STATEMENT.task;
      B := Out_Table_ADT.retrieve(current.comm_list↑.out_list, T, E);
      B↑.info_in.write_try(current.name);
      B↑.info_in.write_wait(current.name);
      B↑.info_in.write(current.name, STATEMENT.in_params);
      return(true, B↑.info_out.read);
    end;
```

Figure 5.6. The "Deterministic_Send" event for Ada.

being called and the name of the task owning the entry are determined from the entry call statement. These are stored in local variables "E" and "T", respectively. The appropriate interface object is then identified by searching the out_list table of the communication list for the task instance containing this entry. Next, the process identification name of the caller is placed on the queue of the interface by the operation "write_try"; the process name in this case is represented by "current.name". Then "write_wait" is called and returns when the task gets to the head of the queue and is available to be used. The sending task then sends its parameters to the shared data object using "write". The "write" operation returns when the called task has accepted the call. After the initial parameters have been passed, the event shifts focus to the "return" operation. The boolean field is set to true, to indicate successful completion of the rendezvous, and the **out** parameters are returned by invoking the "read" operation of the SDA in the info_out field of the interface object.

5.4.3.2 The Conditional_Send Event

A select statement may indicate that the entry call is conditional, as in the following example:

```
select
   T1.E1(PI, PO); ...
else
   ...
end select;
```

If the rendezvous for the entry call "T1.E1(PI, PO)" cannot execute immediately, then the else clause is executed. (Here, "PI" represents the parameters being passed to the called task, while "PO" represents the parameters being passed out after the rendezvous has completed.)

A timed entry call is similar; for example, consider:

```
select
   T2.E1(PI, PO); ...
else
   delay 30.0; ...
end select;
```

In this case, the call "T2.E1(PI, PO)" has 30 seconds in which to execute a rendezvous. If it is unable to do so, the call is not completed and the select statement executes the remainder of the else clause.

The semantics of both of the above kinds of conditional entry call is covered by the event described in Figure 5.7. The event commences by obtaining the name of the entry and the task from the statement, before once again calling the operation

```
Conditional_Send =
    begin
        E := STATEMENT.entry;
        T := STATEMENT.task;
        B := Out_Table_ADT.retrieve(current.comm_list↑.out_list, T, E);
        B↑.info_in.write_try(current.name);
        -- Potential delay occurs here in the case of a timed entry call.
        S := B↑.info_in.can_write_complete?(current.name);
        if S then
            B↑.info_in.write(current.name, STATEMENT.in_params);
            return(true, B↑.info_out.read);
        else
            return(false, null);
        fi;
    end;
```

Figure 5.7. The "Conditional_Send" event for Ada.

"write_try" to notify the shared data object that the task instance is available for communication and placing its process name on the queue. Then, at some point determined by the task (it may be influenced by delay and terminate statements, among others), "can_write_complete?" is sent to the interface object. This can result in one of two outcomes. If the interface indicates that the called task can receive (i.e., meaning that the other task has executed a "read_try" to the interface object), the task can then execute the "write" on the shared data object, which then synchronously transfers the parameters. If the interface object does not have any task which wishes to receive, then the "can_write_complete?" returns "false" and at the same time removes the indication of intention to send (i.e., the "write_try") from the queue. The boolean variable "S" indicates whether the call was made. If the call was made, then the **in** mode parameters are passed to the called task by writing to its info_in port, before the "return" operation is executed. This operation indicates that the rendezvous successfully completed and reads the **out** mode parameters from the info_out port of the called task. If the call was not made (i.e., the boolean variable "S" has the value false), then the event indicates that the rendezvous did not take place and the information returned is "null".

Notice that the "write_try" operation places the process name of the calling task onto the queue, and that the call on "can_write_complete?" returns immediately (but also performs queue management). An important distinction between the deterministic and conditional send statements is the fact that determinism does not represent a "busy wait", because that could cause the task to lose its place in the queue; the waiting occurs within the SDA object.

5.4.3.3 The Deterministic_Receive Event

The simplest type of message reception statement occurs when the accept statement is used deterministically (i.e., not in a select statement). An example of an accept statement is the following:

accept E1(A: **in** a_type; B: **out** b_type) **do** ... **end**;

In the corresponding event, described in Figure 5.8, the name of the entry point is extracted from the statement and stored in the local variable "E". The task then notifies the appropriate SDA that it is able to receive values by calling the operation "read_try". Then the operation "read_wait" is called, returning when there is a sending task available. Finally, "read" is called and returns when the **in** parameters are transferred. The **out** parameters are transferred from the called task to the calling task via the SDA instance in the info_out field. This ensures asynchronous communication, as our interpretation of the ALRM (discussed in Section 5.3) suggests that the rendezvous has completed by the time the called task reaches this point. The "return" operation at the end of the event reflects this by returning the boolean value

```
Deterministic_Receive =
    begin
       E := STATEMENT.entry;
       B := In_Table_ADT.retrieve(current.comm_list↑.in_list, E);
       B↑.info_in.read_try;
       B↑.info_in.read_wait;
       FORMAL_PARAMETERS := B↑.info_in.read;
          -- The body of the accept statement.
       B↑.info_out.write(STATEMENT.out_params);
       return(true, null);
    end;
```

Figure 5.8. The "Deterministic_Receive" event for Ada.

true, indicating that the rendezvous completed successfully. There is no need to wait until the **out** mode parameters have been received by the calling task.

5.4.3.4 The Nondeterministic_Receive Event

The Nondeterministic_Receive event is described in Figure 5.9 and corresponds in some ways to the Conditional_Send event: the nondeterministic selection occurs in the select statement. Consider the following case of nondeterministic message reception:

```
select ...
   accept E1(A: in a_type; B: out b_type) do ... end;
or
   ...
end select;
```

In this case, the presence of a select statement indicates nondeterminism. Also, the accept statement has a "**do** ... **end**" body which is executed if and when the accept statement is executed.

```
Nondeterministic_Receive =
    begin
        E := STATEMENT.entry;
        B := In_Table_ADT.retrieve(current.comm_list↑.in_list, E);
        B↑.info_in.read_try;
        R := B↑.info_in.can_read_complete?;
        if R then
            FORMAL_PARAMETERS := B↑.info_in.read;
            -- The body of the accept statement.
            B↑.info_out.write(STATEMENT.out_params);
        fi;
        return(R, null);
    end;
```

Figure 5.9. The "Nondeterministic_Receive" event for Ada.

There are two types of select statements which can contain accept statements. First, there are *closed select statements*, which have no else clause. They wait until one of the accept statements may execute, at which point it is executed. If no accept statements are ready to execute at the beginning of the execution of the select statement, the statement waits until one is eligible to be executed. On the other hand, and an *open select statement* contains an else clause. In this case, if no accept statements are able to complete, the else clause is executed. If some accept statements are eligible to execute, one is chosen nondeterministically.

In the event shown in Figure 5.9, which describes the effect of nondeterministic message reception in Ada, the boolean variable "R" indicates whether the call was made. The "E" and "B" variables are local and are used to retrieve a reference to the correct interface object. After the reference has been retrieved, the info_in shared data object has the "read_try" operation performed on it. This needs no process name because the task executing the "read_try" is the only task reading from the interface. This notifies the data object that the task is willing to read. The data object is then queried (by the use of the operation "can_read_complete?") as to whether a task is willing to write to the object. If there is a task willing to rendezvous, the variable "R" is set to "true" and the actual parameters are retrieved from the interface by the "read" operation. The body of statements associated with the group is then executed and then the **out** parameters are returned asynchronously via the data object in the info_out field, using the "write" operation. The "return" operation at the end of the event returns the boolean value "R" which indicates whether or not the rendezvous took place. There is no need to wait until the **out** mode parameters have been received by the calling task.

5.4.4 Final Comments on Mallgren's Approach

Mallgren's approach to the definition of shared data abstractions binds together the definition of the data type and its associated operations with a definition of the synchronization aspects important in a concurrent environment. As mentioned earlier, this approach associates a "state" with each SDA object. This state is managed by the SDA definition itself, which does not export a type name; hence, it prevents multiple objects of this kind from being defined.

The style of the SDA definition is somewhat different to that of ADT's and hence the incorporation of the approach described in this section into ATLANTIS would necessitate the reader of a language definition knowing and understanding two, albeit related, but somewhat different, specification techniques for data types. Which approach is used in a particular situation would depend on the manner in which the data object concerned is to be used.

To further complicate matters, an implementation which remains faithful to the foregoing approach is impractical to implement. This stems from the fact that the technique relies on the existence of a potentially infinite history of all the state transitions. Although it is possible to significantly reduce this reliance in the implementation of specific cases, it is not possible to remove this reliance in general.

The characteristic functions employed also present a minor concern. No restrictions are placed on their complexity and as such they may introduce unnecessary serialization.

These concerns are sufficient to suggest that Mallgren's approach to SDA's may not be the most appropriate choice for extension of the technique of Chapters 3 and 4 to cover parallel programming languages, despite the fact that the technique has demonstrated its usefulness in the descriptions of Ada's intertask communication facility above, and in [33]. An alternative approach which fits better with the ATLANTIS approach is discussed in the next section.

5.5 An Alternative Approach to Shared Data Abstractions

Incorporating Mallgren's approach to SDA's into the technique described in Chapters 3 and 4 would force the designer to use ADT's for a sequential environment and SDA's for a parallel environment. This results in unnecessary work on the part of a language designer if some form of concurrency is suddenly introduced into the subject language.

A purely sequential program with only a single process does not have the problems and difficulties that manifest themselves in a parallel program with many processes. There is no need to consider what happens if two processes attempt to access the same data object, there is no notion of synchronization, and often there is no notion of nondeterminism. It is the presence of multiple processes which results in the need

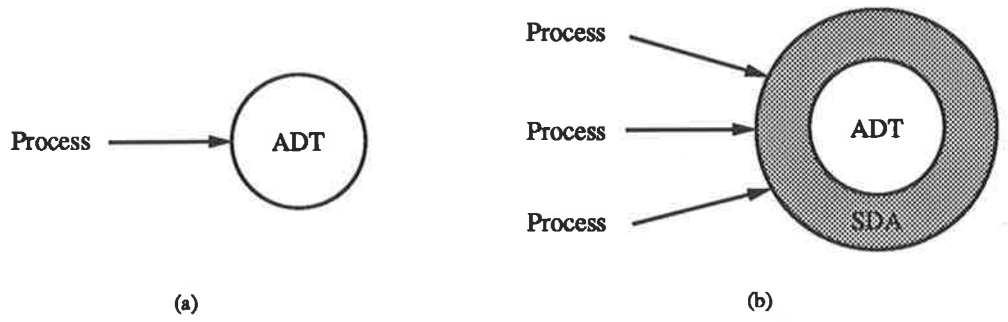


Figure 5.10. Sequential and parallel access to an abstract data type object.

to protect the ADT and ensure data integrity; thus, it can be argued that it is not the responsibility of the ADT definition to cater for the possibility of the existence of multiple processes. By protecting the ADT with an *SDA envelope*, data integrity can be ensured without any need to alter the definition of the data type [107].

The SDA envelope behaves as a protective device for an ADT by mandating that all access to the ADT be handled by the SDA envelope, as illustrated in Figure 5.10(b), and contrasts with the ADT access shown in Figure 5.10(a). Thus, the ADT definition remains unchanged and only concerns itself with the description of the data type and its operations whilst the SDA envelope is concerned with the synchronization issues which arise as multiple simultaneous access becomes a possibility. The SDA envelope then determines which process gains access to the data object.

The attractiveness of the SDA envelope over Mallgren's approach to SDA's is that the ADT definition remains unaltered and objects shared between multiple processes can still be declared. Furthermore, the SDA envelope can be automatically generated from the ADT definition and remain transparent to the user of ATLANTIS. As a result, ATLANTIS need not undergo any changes in order to accommodate the description of parallel languages. The ability to automatically generate the SDA envelope stems from the observation that the operations for each ADT fall into one of the two following categories:

- Selectors: This category comprises those operations which *select* relevant pieces of information from an ADT object and return them, but do not modify the value of an ADT object.

- Constructors: This category comprises those operations which *construct* new ADT objects, whose value may be derived from existing ADT objects.

The following rules are sufficient for an implementation of a shared data abstraction to maintain the integrity of data objects at the level of a single ADT operation:

- At any given time, there may be at most one process performing a constructor operation on an object. This process is given exclusive access to the object for the duration of the operation.

- There may be many selector operations (transfer functions) simultaneously performed on an object, provided that no constructor operations are currently being performed.

These rules ensure that each constructor is treated as though it is an atomic operation. This restriction is unnecessary for selector operations since they do not interfere with each other's behaviour in any way. Constructor operations must be treated as atomic actions since they update the object's associated value. This ensures that no selector operation will accidentally yield an incorrect result.

Figure 5.11(a) illustrates the first of the above rules: *Process A* is performing a constructor operation on an object and is given exclusive access to the object; at the same time, *Process B* and *Process C* are blocked attempting to perform selector and constructor operations on the object, respectively. The second of the above rules is illustrated in Figure 5.11(b): *Process A* and *Process B* are simultaneously performing

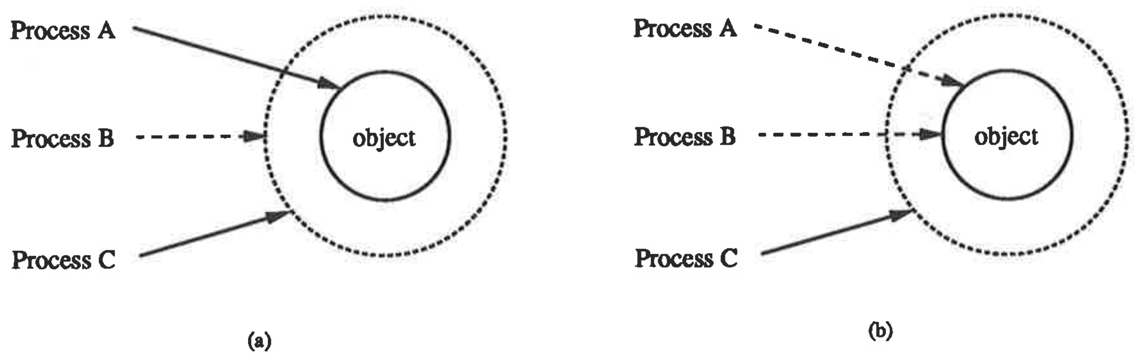


Figure 5.11. An example of the application of the object access rules.

selector operations without mutual interference, whilst *Process C* is blocked waiting for exclusive access to perform a constructor operation.

The rules listed above give no preference to selector or constructor operations, although the SDA envelope may be tailored to favour one over the other; for example, constructors may always be given preference over selectors, or may be given preference only for some percentage of the time (such as giving constructors preference 70% of the time).

Consider the ADT definition of an infinite queue given in Figure 5.12. It can be seen from the definition that the operations "create_queue", "place_on_queue", "remove_head" and "remove_entry" are all constructors, whilst the selector operations are "head_of_queue", "present_in_queue", "queue_empty?" and "length_of_queue".

As discussed in Chapter 4, ADT's defined in an axiomatic manner can readily be translated into an implementation using a language facility such as Ada packages. The generic Ada package corresponding to Figure 5.12 is provided in Figure 5.13. By comparing these two figures, it can be seen that each operation specified in Figure 5.12 has a corresponding function, set of arguments and result type in Figure 5.13. The exception "queue_error" is introduced into the Ada package to implement the "error" term used in the formal ADT specification. Since the ADT specified in Figure 5.12 is

```
ADT queue [elem]
sorts queue/elem, boolean, natural
where elem has equal: elem × elem → boolean

syntax
   create_queue:                       →   queue
   place_on_queue:    queue × elem     →   queue
   remove_head:       queue            →   queue ⋃ {error}
   head_of_queue:     queue            →   elem ⋃ {error}
   present_in_queue:  queue × elem     →   boolean
   remove_entry:      queue × elem     →   queue ⋃ {error}
   queue_empty?:      queue            →   boolean
   length_of_queue:   queue            →   natural

semantics
   declare q: queue
           e1, e2: elem

axioms
   (1)  remove_head(create_queue) = error
   (2)  remove_head(place_on_queue(q, e1)) = q
   (3)  head_of_queue(create_queue) = error
   (4)  head_of_queue(place_on_queue(q, e1)) = e1
   (5)  present_in_queue(create_queue, e1) = false
   (6)  present_in_queue(place_on_queue(q, e1), e2) =
            if equal(e1, e2)
            then
               true
            else
               present_in_queue(q, e2)
            end if
   (7)  remove_entry(create_queue, e1) = error
```

Figure 5.12. An abstract data type representing a queue.

```
(8)  remove_entry(place_on_queue(q, e1), e2) =
          if equal(e1, e2)
          then
             q
          else
             place_on_queue(remove_entry(q, e2), e1)
          end if
(9)  queue_empty?(create_queue) = true
(10) queue_empty?(place_on_queue(q, e1)) = false
(11) length_of_queue(create_queue) = 0
(12) length_of_queue(place_on_queue(q, e1)) =
          length_of_queue(q) + 1
```

Figure 5.12. Continued.

```
generic
   type elem is private;
   with function equal(e1, e2: in elem) return boolean;
   with function copy(e1: in elem) return elem;

package queue_adt is
   type queue is private;

   function create_queue return queue;
      -- Create a new queue object and return it.

   function place_on_queue(q1: in queue;
                           e1: in elem) return queue;
      -- Place the specified element at the end of the queue.

   function remove_head(q1: in queue) return queue;
      -- Remove the element at the head of the queue and return the
      -- resultant queue.
```

Figure 5.13. The specification of the queue data type.

```
  function head_of_queue(q1: in queue) return elem;
     -- Return the element at the head of the queue, leaving the queue
     -- unchanged.

  function present_in_queue(q1: in queue;
                                   e1: in elem) return boolean;
     -- Is the specified element on the queue?

  function remove_entry(q1: in queue; e1: in elem) return queue;
     -- Remove the specified element from the queue and return the
     -- resultant queue.

  function queue_empty(q1: in queue) return boolean;
     -- Is the queue empty?

  function length_of_queue(q1: in queue) return natural;
     -- Return the number of elements currently in the queue.

  function copy(q1: in queue) return queue;
     -- Duplicate the queue and return the copy.

  queue_error: exception;
     -- For the handling of errors.

private
  type queue is ...;
     -- Queue can be defined in any suitable way by the implementor
     -- of the package.

end queue_adt;
```

Figure 5.13. Continued.

parameterized, the corresponding package implementation is generic with the actual type (and function) parameters provided at the time that the package is instantiated. One additional operation is assumed in the package specification, despite the fact that it is not specified by the ADT definition. This is the existence of a "copy" operation for each data type as discussed in Section 4.7. The signature of the ADT specification and the Ada package specification describe the same data type and the operations available on it, while the package body represents an implementation which adheres to the behaviour described by the axioms of the ADT.

The SDA envelope generated for the ADT provided by Figure 5.13 is given in Figures 5.14 and 5.15. As can be seen, the SDA envelope is essentially another package which provides the same operations as the ADT, with the SDA envelope body invoking the ADT operations. Protection of the ADT object is achieved by a monitor [61] which enforces the rules for SDA integrity discussed earlier.

It can be seen from Figures 5.14 and 5.15 that all constructor operations have been converted to procedures whose first parameter is the object to be returned. This may at first suggest that the user of ATLANTIS must be aware of this difference and define a programming language accordingly. In reality, however, the user is unaware of the change. In fact, the user of ATLANTIS can remain ignorant of the existence of the SDA envelope. This is because ATLANTIS itself can generate the SDA envelope to protect ADT's when defining a parallel language, and ATLANTIS can translate all calls to ADT constructor functions to calls to the SDA constructor procedures of the SDA envelope.

```
with queue_package, sda_monitor;  use sda_monitor;
pragma elaborate(sda_monitor);

generic
  type elem is private;
  with function equal(e1, e2: in elem) return boolean;
  with function copy(elem: in data) return elem is <>;

package sda_queue is

  type queue is private;
  type queue_access is private;

  procedure create_queue(result: in out queue_access);
    -- Create a queue.

  procedure place_on_queue(result: in out queue_access;
                           q1: in queue_access;
                           e1: in elem);
    -- Push some data e1 onto the end of the queue q1.

  procedure remove_head(result: in out queue_access;
                        q1: in queue_access);
    -- Remove the element at the head of the queue q1.

  function head_of_queue(q1: in queue_access) return elem;
    -- Return the value at the head of the queue q1.

  function present_in_queue(q1: in queue_access;
                            e1: in elem) return boolean;
    -- Indicate if the element e1 is a member of the queue q1.

  procedure remove_entry(result: in out queue_access;
                         q1: in queue_access;
                         e1: in elem);
    -- Remove the entry e1 from the queue q1.

  function queue_empty(q1: in queue_access) return boolean;
    -- Test to see if the queue q1 is empty.
```

Figure 5.14. The SDA envelope specification for the queue data type.

```
  function length_of_queue(q1: in queue_access) return natural;
    -- Return the length of the queue q1.

  function new_shared_object return queue;
    -- Initialize the SDA queue object.

  procedure lock(queue: in queue_access);
    -- Locks an object for use by a single process.

  procedure unlock(queue: in queue_access);
    -- Unlocks an object previously locked.

  function access_SDA(queue: in queue) return queue_access;
    -- Establish the SDA object.

  package queue_inst is new queue_package(elem, equal, copy);
    -- Instantiate the queue ADT.

  queue_error: exception renames queue_inst.queue_exception;
    -- Error raised if anything goes wrong.

private

  type queue_record_type is
    record
      object: queue_inst.queue_type;
      Monitor: monitor_type := new_monitor;
      -- "monitor_type" and "new_monitor" are defined within the
      -- package "sda_monitor".
    end record;

  type queue_type is access queue_record_type;
```

Figure 5.14. Continued.

```
  type queue_access_record_type is
    record
       sda: queue;
       mode: access_rights_type := default_access;
       -- "access_rights_type" and "default_access" are defined within
       -- the package "sda_monitor".
    end record;

  type queue_access is access queue_access_record_type;

end sda_queue;
```

Figure 5.14. Continued.

```
package body sda_queue is

  function new_shared_object return queue_type is
     -- Initialize the SDA queue object.
  begin -- new_shared_object
    return new queue_record_type;
  end new_shared_object;

  function access_SDA(q1: in queue_type) return queue_access is
    -- Establish the SDA object.
  begin -- access_SDA
    return new queue_access_record_type'(q1, default_access);
  end access_SDA;

  procedure lock(q1: in queue_access) is
      -- Locks an object for by a single process.
  begin -- lock
    lock(q1.sda.monitor, q1.mode);
  end lock;
```

Figure 5.15. The package body for the shared data abstraction envelope for the queue abstract data type.

```
procedure unlock(q1: in queue_access) is
    -- Unlocks an object previously locked.
begin -- unlock
   unlock(q1.sda.monitor, q1.mode);
end unlock;

procedure create_queue(result: in out queue_access) is
   -- Create a queue.
begin -- create_queue
   start_constructor(result.sda.monitor, result.mode);
   result.sda.object := queue_inst.make_queue;
   stop_constructor(result.sda.monitor, result.mode);
end create_queue;

procedure place_on_queue(result: in out queue_access;
                         q1: in queue_access;
                         e1: in elem) is
   -- Push some data onto the end of the queue q1.
begin -- place_on_queue
   start_constructor(result.sda.monitor, result.mode);
   start_selector(q1.sda.monitor, queue.mode);
   result.sda.object := queue_inst.place_on_queue(q1.sda.object, elem);
   stop_selector(q1.sda.monitor);
   stop_constructor(result.sda.monitor, result.mode);
end place_on_queue;

procedure remove_head(result: in out queue_access;
                      q1: in queue_access) is
   -- Remove the element at the head of the queue q1.
begin -- remove_head
   start_constructor(result.sda.monitor, result.mode);
   start_selector(q1.sda.monitor, q1.mode);
   result.sda.object := queue_inst.remove_head(q1.sda.object);
   stop_selector(q1.sda.monitor);
   stop_constructor(result.sda.monitor,result.mode);
end remove_head;
```

Figure 5.15. Continued.

```
function head_of_queue(q1: in queue_access) return elem is
   -- Return the value at the head of the queue q1.
   result: elem;
begin -- head_of_queue
   start_selector(q1.sda.Monitor, q1.mode);
   result := queue_inst.head_of_queue(q1.sda.object);
   stop_selector(q1.sda.monitor);
  return result;
end head_of_queue;

function present_in_queue(q1: in queue_access;
                          e1: in elem) return boolean is
   -- Indicate if the element e1 is a member of the queue q1.
   result: boolean;
begin – present_in_queue
   start_selector(q1.sda.monitor, q1.mode);
   result := queue_inst.present_in_queue(q1.sda.object, e1);
   stop_selector(q1.sda.monitor);
  return result;
end present_in_queue;

procedure remove_entry(result: in out queue_access;
                       q1: in queue_access;
                       e1: in elem) is
   -- Remove the entry e1 from the queue q1.
begin -- remove_entry
   start_constructor(result.sda.monitor, result.mode);
   start_selector(q1.sda.monitor, q1.mode);
   result.sda.object := queue_inst.remove_entry(q1.sda.object, e1);
   stop_selector(q1.sda.monitor);
   stop_constructor(result.sda.monitor, result.mode);
end remove_entry;
```

Figure 5.15. Continued.

```
   function queue_empty(q1: in queue_access) return boolean is
      -- Test to see if the queue q1 is empty.
      result: boolean;
   begin -- queue_empty
      start_selector(q1.sda.Monitor, q1.mode);
      result := queue_inst.empty_queue(q1.sda.object);
      stop_selector(queue.sda.monitor);
     return result;
   end queue_empty;

   function length_of_queue(q1: in queue_access) return natural is
      -- Return the length of the queue q1.
      result: natural;
   begin -- length_of_queue
      start_selector(q1.sda.monitor, q1.mode);
      result := queue_inst.length_of_queue(q1.sda.object);
      stop_selector(q1.sda.monitor);
      return result;
   end length_of_queue;

end sda_queue;
```

Figure 5.15. Continued.

As noted above, procedures are used within the SDA envelope in order to properly protect the SDA object. Had a functional notation and the notion of copying used for ADT's been retained, then a statement such as:

a := constructor_op(a);

would result in a copy being made of the object "a", the constructor operation applied to the copy and the result assigned to "a". Under these circumstances, there is no need to protect the object "a" and concurrent access is of no concern, simply because there is no way to guarantee the desired behaviour and results.

Examination of Figure 5.14 shows that the SDA envelope makes use of a package called "sda_monitor". This package is employed by all generated SDA envelopes and its specification is given in Figure 5.16.

```
package sda_monitor is
   type monitor_type is private;
   type access_rights_type is private;
   default_access: constant access_rights_type;

   procedure start_constructor(monitor: in monitor_type;
                               access_rights: in out access_rights_type);
     -- Allows the process which successfully executes this operation to
     -- apply a constructor operation to the SDA object associated with
     -- the monitor.

   procedure start_selector(monitor: in monitor_type;
                            access_rights: in out access_rights_type);
     -- Allows the process which successfully executes this operation to
     -- apply a selector operation to the SDA object associated with
     -- the monitor.

   procedure stop_constructor(monitor: in monitor_type;
                              access_rights: in out access_rights_type);
     -- Informs the monitor that the constructor operation which was to
     -- be applied to the SDA object associated with the monitor has
     -- been completed.

   procedure stop_selector(monitor: in monitor_type);
     -- Informs the monitor that the selector operation which was to be
     -- applied to the SDA object associated with the monitor has been
     -- completed.

   procedure lock(monitor: in monitor_type;
                  access_rights: in out access_rights_type);
     -- Applies a lock to the SDA object associated with this monitor so
     -- that the process holding the lock is the only process permitted
     -- to apply any constructor operation to the object.

   procedure unlock(monitor: in monitor_type;
                    access_rights: in out access_rights_type);
     -- Releases the lock on the SDA object associated with the monitor.

   function new_monitor return monitor_type;
     -- Creates a new monitor.
```

Figure 5.16. The specification of the "sda_monitor" package.

```
private
  type key_type is (GOOD_KEY,NO_KEY);
  type access_rights_type is
    record
      construct_select_access: key_type := no_key;
      lock_access            : key_type := no_key;
    end record;

  task type monitor_task is
    entry constructor_start(key_type);
    entry selector_start(key_type);
    entry constructor_stop;
    entry selector_stop;
    entry lock;
    entry unlock;
  end monitor_task;

  type monitor_type is access monitor_task;
  default_access: constant access_rights_type
                         := access_rights_type'(no_key, no_key);
end sda_monitor;
```

Figure 5.16. Continued.

The "sda_monitor" package ensures that no two constructor operations simultaneously manipulate the SDA object and that no constructor operation is permitted to commence whilst the execution of a selector operation on the object is in progress. It also allows multiple selector operations to simultaneously access an SDA object, but prevents a selector operation from commencing if a constructor operation is in progress. These restrictions are enforced by the application of the operations "start_constructor" and "stop_constructor", respectively, before and after the application of a constructor operation. Similarly, the selector operations are protected by the operations "start_selector" and "stop_selector" being applied before and after the application of a selector operation, respectively. The operations "lock" and "unlock" are provided to allow a process to place a lock on an SDA object in a manner that only permits that process to execute constructor operations on the SDA object until the lock is released. All other processes wishing to apply a constructor operation to the SDA object while a lock is in place are forced to wait until the lock is released. Selector operations can, however, continue to be applied to a locked object without any adverse affect. This locking concept is discussed further in Section 5.5.5.

Figure 5.16 makes use of a type named "access_rights_type". This type governs which process has the right to apply a constructor operation, or a selector operation, and which process, if any, holds the lock on the object. Each process presents its access rights to the monitor associated with with the SDA object whenever the process wishes to perform an operation on the SDA object. The access rights are checked to ensure that the rules governing access to SDA objects are not violated. If the "lock" operation has been invoked, then the calling process will have its "lock_access" field changed to "good_key" and all subsequent constructor operations must have access rights such that the "lock_access" field has this value. Similarly, when an operation executes a "start_constructor" operation, its "construct_select_access" field is set to "good_key". All subsequent selector operations, which must call "start_selector" before proceeding, are blocked unless they have this value in this field, indicating that this is the same process wishing to apply a selector operation to the SDA object as part of the normal work involved in evaluating a constructor operation which is in progress. The appropriate fields assume the default value when the lock is released via the "unlock" operation, or if the constructor operation completes (signalled by the execution of a "stop_constructor" operation). Since the SDA envelope is generated automatically, one can be sure that each call to "start_selector" or "start_constructor" is balanced by a call to "stop_selector" or "stop_constructor" respectively, ensuring that the objects' access rights are maintained in a consistent state.

5.5.1 Using Shared Data Abstractions

As an example of the use of an SDA generated in the manner described above, consider employing the queue SDA used as an example above. In particular, consider a producer process placing integers onto such a queue and a consumer process removing integers from the queue. First, the "sda_queue" package is instantiated, using "integer" as the type corresponding to the generic parameter "elem"; an "equal" operation for objects of type integer and a "copy" routine to duplicate integer values are also passed as generic parameters. The appropriate declarations are:

```
package queue_sda is new sda_queue(integer, equal, copy);
use queue_sda;
```

A queue object can then be obtained by declaring a variable of the type "queue" belonging to the SDA type "queue_sda", as follows:

```
the_queue_sda: queue;
```

The task specifications and bodies for the producer and consumer processes are shown in Figure 5.17. In the case of a single producer and a single consumer, the queue behaves correctly and all information is placed onto the queue by the producer and eventually received by the consumer. In the case of multiple producers and a single consumer, the queue still handles the traffic correctly and all integer values are eventually passed to the consumer for processing; although multiple producers may attempt to simultaneously push a value onto the queue, the monitor implemented by the SDA envelope arbitrates correctly. The "place_on_queue" operator is a constructor operator and only one constructor operator may manipulate the SDA object at any one time, provided that no selector operator is active at the same time. This ensures

```
task type producer;
task type consumer;

 the_queue: queue_access := access_sda(the_queue_sda);

task body producer is
begin
  loop
     place_on_queue(the_queue, the_queue, i);
    delay duration'small;
  end loop;
end producer;

task body consumer is
begin
  loop
    if not queue_empty(the_queue)
    then
      i := head_of_queue(the_queue);
      remove_head(the_queue, the_queue);
      consume(i);
    end if;
  end loop;
end consumer;
```

Figure 5.17. Example of the use of a shared data abstraction.

that the producers attempting simultaneous "place_on_queue" operations have some ordering imposed on them.

However, a difficulty is encountered in the case of multiple coexisting consumers. Each consumer task copies the head of the queue into a local variable before deleting that element from the queue. It is then possible that two consumers simultaneously copy the object at the head of the queue into two different variables. This is permitted because the operation "head_of_queue" is a selector and since it does not alter the SDA object in any way; hence, it is valid for multiple processes to execute it simultaneously. This is permitted even if the queue contains only a single element. These consumers then both attempt to remove the value at the head of the queue. Since this is a constructor operation, an ordering is imposed on the attempts to execute the operation. The first invocation of the operation removes the head of the queue, whilst subsequent invocations attempt to remove as yet unread values from the queue, or values that do not exist. Clearly, this is an undesirable result. The difficulty is, however, in the algorithm used to implement a consumer, rather than in the behaviour of the SDA. The SDA has accommodated the maximum amount of concurrency, but the algorithm has made no attempt to acknowledge that multiple accesses may take place, leaving all the details to the SDA to handle.

These difficulties may be overcome through traditional methods, such as the use of *semaphores* [27] to provide some protection to the combination of operations "head_of_queue" and "remove_head", so that their invocation becomes a single indivisible action; alternatively, a locking mechanism such as that provided by the package "sda_monitor" may be employed. If the data structure in question is no longer a queue, but rather a stack, then the problems are somewhat worse. With a stack, it is even possible to have problems with a single producer and a single consumer. The producer may add a value to the stack. The consumer may then note that the stack is no longer empty and copy the top of the stack into a local variable in preparation for the removal of the value from the stack. If the producer now inserts another element to the stack before the consumer removes the element it has just made a copy of, then when the consumer deletes the top element of the stack, it does so believing that it is the element that it just copied. The result is the removal of a value which has never been seen by a consumer and the retention of a value which will be processed twice. Again, this is an undesirable state of affairs, and again the problem is an ill-defined algorithm for the behaviour of the consumer.

To aid language designers overcome these difficulties with their language definitions within ATLANTIS, a number of language constructs are proposed in the following sections. These constructs allow the clear, precise and unambiguous description of the algorithms needed to manipulate SDA objects, by solving the difficulties described above.

5.5.2 Critical Regions

One way to specify the concurrency control referred to above is to use the notion of a *critical region* [27]. A critical region amounts to a section of a program specification in which only one process may be active at any one time. Such constructs have traditionally been used in programming languages, but are equally useful in the present context. Our notation to define a critical region is shown in Figure 5.18 and can be defined in terms of semaphores, as shown in Figure 5.19. A diagrammatic representation of a critical region is shown in Figure 5.20; the shaded area represents the region in which only one process may be active at any one time. The effect of a critical region is to ensure mutual exclusion for a region of the specification; each area of mutual exclusion will have its own semaphore. Any processes attempting to gain access to an area of mutual exclusion is forced to wait until the appropriate semaphore variable has been released by the currently executing process. On release of the semaphore, a nondeterministic choice is made between each of the waiting processes and the chosen process is permitted to continue. Alternative definitions are possible, for example in terms of denotational semantics; however, the semaphore concept is sufficient for our purposes and is well established and well understood.

```
**critical**
    S1; ...; Sn;
**end critical;**
```

Figure 5.18. The textual form of a critical region.

```
wait(v);
S1; ...; Sn;
signal(v);
```

Figure 5.19. Definition of a critical region in terms of semaphores.

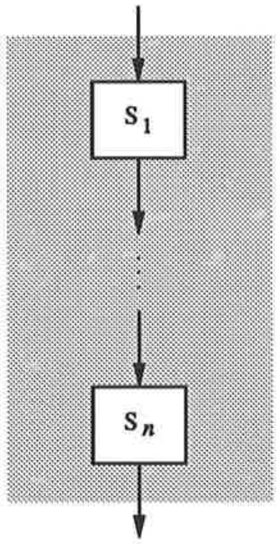


Figure 5.20. A diagrammatic representation of a critical region.

5.5.3 The Waituntil Clause

When programming with multiple processes, there is a need to synchronize events at certain times during the lifetime of a program; this is particularly true in languages such as Ada [141] and Modula-2 [159]. To model this situation, the *waituntil clause* is introduced; its syntax is given in Figure 5.21.

```
waituntil cond →
    $S_1$; ...; $S_n$;
end waituntil;
```

Figure 5.21. The textual form of a waituntil clause.

```
loop
    wait(v);
    if cond then
        $S_1$; ...; $S_n$;
        signal(v);
        exit;
    else
        signal(v);
    end if;
end loop;
```

Figure 5.22. Definition of the waituntil clause using semaphores.

The semantics of this clause is defined in terms of semaphores in Figure 5.22 and its diagrammatic form is shown in Figure 5.23. The effect of a waituntil clause is to evaluate the condition "cond" under mutual exclusion with regard to any other processes, using the semaphore variable "v" to achieve this mutual exclusion. If the condition evaluates to true, the statement sequence is executed while maintaining mutual exclusion; if the condition evaluates to false, the semaphore variable is released and then the mutual exclusion is reimposed and the condition is re-evaluated once again, possibly by a different process, thus implementing a busy wait. The release of the semaphore ensures that a nondeterministic choice is made between all available processes waiting on the semaphore.

Thus, the boolean condition must be satisfied before the evaluation may proceed. If the condition is satisfied, then the statement list which follows is treated as a critical region until the **end waituntil** is reached; if the condition is not satisfied, then the

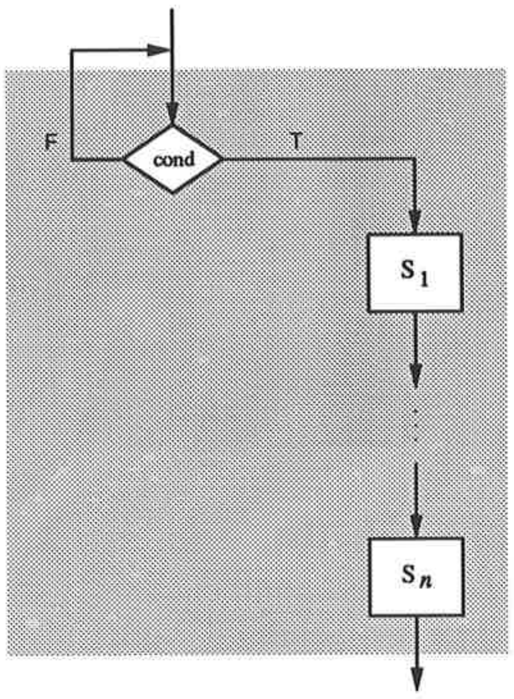


Figure 5.23. A diagram representing the waituntil clause.

process executing this construct is blocked until the condition becomes true. If more than one process is waiting at the condition, then a nondeterministic choice is made to determine which process is allowed to enter the critical region.

5.5.4 The Do Clause

Often a process may reach a point where several options are open to it. The process is not concerned with which path it takes; all it must do is simply choose one of the available paths. The *do clause* is introduced to handle nondeterminism between a variety of choices, each specified by a *when clause*. An outline of the syntax of this clause is given in Figure 5.24. This clause is equivalent to the pseudo–Ada code shown in Figure 5.25; in this definition of the do clause, the function "choice" yields, in a nondeterministic fashion, an integer between 1 and the value of the argument ("m" in the case of Figure 5.25).

```
do
    when cond$_1$ → S$_{11}$; ...; S$_{1n}$; end when; S$_{1s}$; ... S$_{1r}$;
    ...
    when cond$_m$ → S$_{m1}$; ...; S$_{mn}$; end when; S$_{ms}$; ... S$_{mr}$;
end do;
```

Figure 5.24. The textual form of a do clause.

The do clause is shown diagrammatically in Figure 5.26 and includes the replication of a structure which is somewhat like the depiction of the waituntil clause in Figure 5.23. Once a particular alternative in a do clause is chosen, the appropriate condition is evaluated under the control of a semaphore. If the condition evaluates to true, the first action sequence ("S_{i1}" to "S_{in}") is evaluated as a critical region, the semaphore is then released and the second action sequence ("S_{is}" to "S_{ir}") is evaluated. If the condition evaluates to false, the semaphore is released and a nondeterministic choice of the alternatives is made again.

Thus, more than one process may be active in a do clause at any one time; the only restriction is that there can only be one process active within a given when clause inside a do clause. The do clause does not terminate until at least one of the conditions becomes true and the relevant action sequence has been executed.

5.5.5 The Lock Clause

On occasions it is desirable that a particular task be allowed exclusive write access to a particular object. Tasks that do not modify the object should be allowed to access the value of the object as often as necessary, since such access does not endanger data integrity. To illustrate the need for object locking, consider a task that executes the following code:

```
loop
    case choice(m) is
        when 1 =>
            wait($v_1$);
            if cond$_1$ then
                $S_{11}$; ...; $S_{1n}$;
                signal($v_1$);
                $S_{1s}$; ...; $S_{1r}$;
                exit;
            else
                signal($v_1$);
            end if;
        when 2 =>
            ...
        when m =>
            wait($v_m$);
            if cond$_m$ then
                $S_{m1}$; ...; $S_{mn}$;
                signal($v_m$);
                $S_{ms}$; ...; $S_{mr}$;
                exit;
            else
                signal($v_m$);
            end if;
        when others =>
            null;
    end case;
end loop;
```

Figure 5.25. Definition of the do clause using semaphores.

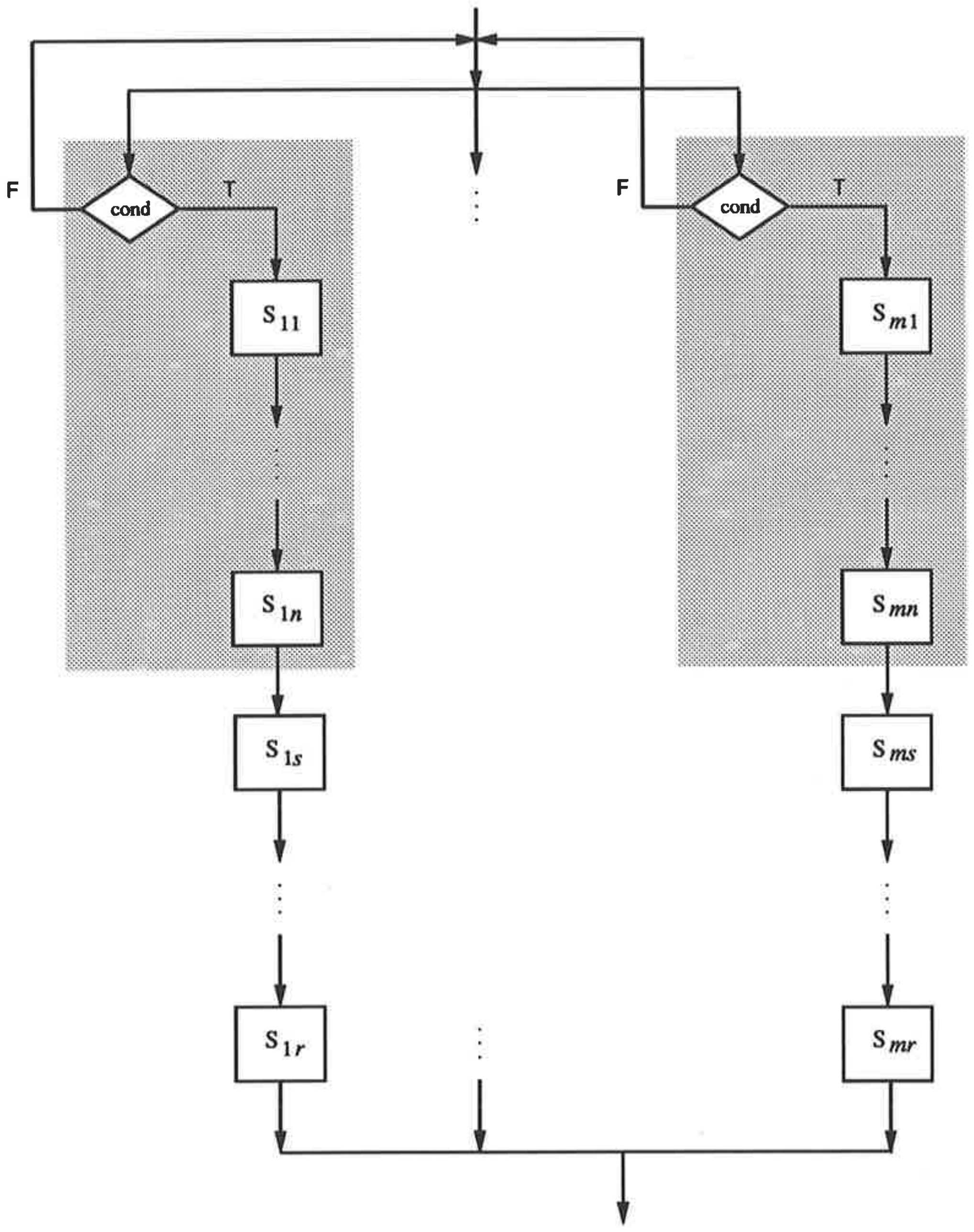


Figure 5.26. A diagrammatic view of the do clause.

```
element := Top(S);   -- get the top value of the stack.
Pop(S);              -- pop the element off the stack.
```

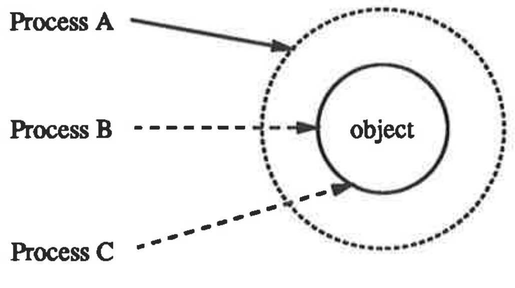
Now suppose that some other task pushes a value onto the stack after the execution of the first statement and before the execution of the second statement. Such an occurrence would result in the above code removing the newly pushed element from the stack, whilst the element that should have been removed is left at the top of the stack.

From the above example, it can be seen that a simple object locking mechanism would provide a safe and effective mechanism for describing indivisible operations, without the tiresome alternative of using critical regions. In essence, a critical region admits exclusive access to a region of code, whilst a lock allows a process to have exclusive access to an object with respect to constructor operators.

Figure 5.27(a) shows the syntax for such locked regions. Here the object is locked by the executing process whilst the statements S_1,..,S_n are executed. This has the effect that any other process attempting to modify the object while it remains locked, will be blocked until such time as the object becomes unlocked. The usual restrictions apply to those processes attempting only to access (not to modify) the object, in that they must wait until the process owning the lock is not performing any *constructor* operations, and vice-versa. Naturally, it is advisable that the number of statements inside a locked region be kept to a minimum, in order to reduce blocking. It is desirable that all code involved in a locked region be as localized as possible. In particular, the invocation of HLO's from within locked regions should also be kept to a minimum, as it detracts from the readability of the specification and increases the possibility of unnecessary delays for other processes.

```
lock (object)
      S1; ...; Sn;
end lock;
```

(a)



(b)

Figure 5.27. Syntactic and diagrammatic representation of locked regions.

The nature of the locking mechanism is further illustrated in Figure 5.27(b). In the example shown, there are three processes “A”, “B” and “C” that each wish to access an object concurrently. Process “C” has placed a lock on the object and is currently performing a *selector* operation on the object. Process “B” also wishes to perform a *selector* operation on the object, and thus is also given access to the object. Process “A” wishes to perform a *constructor* operation on the object, and is blocked. Even when the *selector* operations being performed by processes “B” and “C” have finished, Process “A” will remain blocked until Process “C” relinquishes the lock that it holds on the object and all *selector* operations being performed on the object also finish.

5.5.6 General Comments

As a result of critical regions, the **waituntil** clause and the **do** clause, there is a clear distinction between the definition of the data type and its behaviour and the manner in which it is accessed. Another advantage of this approach is that the three constructs can be related to the relevant features of Ada and other parallel languages, and as a result are likely to be reasonably familiar to programmers. This is particularly useful when the technique is used to produce an operational model of the semantics of a programming language in order to derive a formal definition. The subsequent definition can be used to automatically generate a compiler or interpreter, with all the attendant benefits discussed earlier in this thesis.

5.6 An Alternative Model for the Ada Rendezvous

A model of inter-task communication in Ada using Mallgren's approach to SDA's was presented in Section 5.4. To illustrate the usefulness of the approach described in Section 5.5 and to provide examples of the usage of the constructs given in that section, an alternative model of the inter-task communication in Ada will now be presented.

Each task instance in an Ada program is modelled by a process; the main program instance is also modelled by a process. A unique name is assigned to each of these processes, enabling one process to inform another that it is no longer suspended; for example, this mechanism can be used by a called task to inform its calling task that their rendezvous is complete. A task table is used to manage the various processes in existence at any particular time; each entry in this table is of the type "task_object" defined in Figure 5.28. The first field in this record type is used to record the *communications port* assigned to each process. Such a port may be in one of two states, namely "ready_port" or "busy_port". Only a given process and the process with which it is communicating should change the state of the communication port; this port signals information such as the fact that the transfer of information during a rendezvous has completed. The second field in the record type in Figure 5.28 represents the collection of entry points which may be involved in a rendezvous between this task and other tasks; the type of this field is an array indexed by the primitive type "entry_name_type", representing the set of all allowed names of entries within tasks. The task also has an associated list of attributes (such as "terminated"), which records information pertinent to the task; these attributes will not be described in the simplified model given here.

```
type task_object is
    record
        port: communication_port;
        entry: table [entry_name_type, entry_object];
        -- Attributes can also be described here.
    end record;
```

Figure 5.28. A record describing each Ada task.

```
type entry_object is
    record
        port: communication_port;
        params: parameter_info;
        process_queue: queue[task_name_type];
        -- Attributes can also be described here.
    end record;
```

Figure 5.29. Information associated with each entry point within a task.

Each entry point within a task is described by the type "entry_object" defined in Figure 5.29. Each entry point has a communication port, represented by the field "port", through which it can be signalled when parameter values are available. These parameter values are passed by the calling process placing the necessary details into a table "params" associated with each entry point. This table is custom-built for each entry point and its form relates to the types of the parameters and their transmission modes; it is represented by an object of the primitive type "parameter_info". The relevant information is held in a table indexed by both the position and the name of the parameter (since Ada allows both the positional and named notation for parameter transmission); this information includes: transmission mode, type, initial value and value transmitted.

The third field of the record type in Figure 5.29 represents a queue of task names with which this task entry might rendezvous. This queue is indexed by the set of acceptable task names represented as an object of the primitive type "task_name_type". (The priority of a process is ignored in the present model.) Each entry point also has an associated list of attributes; an example of such an attribute is "count", which records how many tasks are queued waiting to communicate with this task via this entry point. These attributes are not described in the model presented here in order to keep the complexity to a minimum and to allow fair comparison with the model of the rendezvous mechanism employing Mallgren's approach to SDA's presented earlier.

```
task T1 is
   entry e1;
   entry e2;
end T1;

task T2;
   entry e3;
end T2;
```

Figure 5.30. An Ada program fragment.

The information structures employed by this model can be illustrated by considering the Ada program fragment in Figure 5.30. This fragment involves two tasks "T1" and "T2"; the first of these has two entries ("e1" and "e2") and the second has one entry ("e3"). Each task instance will be represented by an object of type task_object, which will associate a communication port and an array of objects of type entry_object with each task. For task "T1", the array of objects of type entry_object will consist of two such objects (one for each entry point) and task "T2" will have an array of only one object. Each entry point will be represented by a record containing its own communication port and some storage for the parameter information. Any attributes relevant to entry points may also be associated with this object of type entry_object.

In order to describe Ada's tasking facility, some primitives are introduced. These primitives are outlined below:

- The identifier "current" will be used to refer to the task name of the current process. It is an object of type "task_name_type".
- The primitive "task_table", already mentioned, is an array of objects indexed by "task_name_type"; each component in the array is of type "task_object".
- The function "timeout_expired" returns a boolean value indicating whether the timeout period for the rendezvous has yet expired.
- The function "actual_params" returns the actual parameter values associated with a call on an entry point.
- The primitive "execute_statements" executes the statements specified by its argument. The valid values for this argument include:
  - "rendezvous" which refers to the statements, if any, which comprise the rendezvous being considered, and
  - "alternative" which refers to the statements, if any, which form the alternative part of a select statement.

Since we are concentrating here on certain aspects of Ada's intertask communication, other aspects of Ada semantics (such as the actual transmission of the parameter information) are merely modelled by the invocation of a HLO and a brief informal description.

5.6.1 Abstract Data Types for Modelling the Rendezvous Mechanism of Ada

The basic ADTs required by the model are the queue ADT given earlier in Figure 5.12 and the communication port ADT specified in Figure 5.31. A third basic ADT employed by the model is a table type, already used in Figure 5.28; its specification is similar to that of the queue presented in Figure 5.12. The table ADT is parameterized with respect to its index type and the type of the elements of the table. The operations "add_to_table" and "retrieve_from_table" are provided to add and retrieve information from a table object.

```
ADT communication_port
sorts communication_port/boolean

syntax
   ready_port:                              →  communication_port
   busy_port:                               →  communication_port
   ready?:       communication_port  →  boolean

semantics
axioms
   (1)   ready?(ready_port) = true
   (2)   ready?(busy_port) = false
```

Figure 5.31. The definition of a communication port.

5.6.2 Deterministic Send

The Deterministic_Send event, discussed in Section 5.4.3.1, may be described using the alternative SDA approach as shown in Figure 5.32.

```
Deterministic_Send(called_task: task_name_type; called_entry: entry_name_type) =
        using
          called_task_object, calling_task_object: task_object;
          called_entry_object: entry_object;
          Q: queue[task_name_type];
        begin
          called_task_object := task_table(called_task);
          called_entry_object := retrieve_from_table(called_task_object.entry,
                                                          called_entry);
          Q := called_entry_object.process_queue;
          calling_task_object := task_table(current);
          -- Place the current task on the queue for entry "called_entry"
          -- of task "called_task".
          Q := place_on_queue(Q, current);
          -- Wait for signal from called task to indicate that it can proceed.
          waituntil ready?(calling_task_object.port) →
            calling_task_object.port := busy_port;
          end waituntil;
          -- Pass parameters.
          called_entry_object.params := actual_params;
          -- Signal called task that parameter transmission is complete.
          called_entry_object.port := ready_port;
          -- Wait for rendezvous to complete.
          waituntil ready?(calling_task_object.port) →
            calling_task_object.port := busy_port;
          end waituntil;
        end;
```

Figure 5.32. The deterministic send event.

Figure 5.32 indicates that the description of the Deterministic_Send event depends on two parameters: the name of the task "called_task" which is called and the name of the entry "called_entry" required within that task. Declarations of local identifiers follows in the section introduced by the keyword **"using"**; their purpose is to simplify the description by having these identifiers stand for longer and more complex expressions. In particular, these local identifiers usually stand for components of the information structures used in the model of the Ada rendezvous. Events such as the Deterministic_Send event of Figure 5.32 are instantiated when the relevant language feature is encountered; in the case of the following entry call on the entry "E1" of the task "T1":

T1.E1;

the arguments "T1" and "E1" are passed to the Deterministic_Send event on instantiation.

The definition of the semantics of the Deterministic_Send event is delimited by the keywords **begin** and **end**. This event commences by indexing the task_table with the name of the invoked task and associates the resulting task_object object with the local variable "called_task_object". The array of entry points for this task is then indexed by the name of the desired entry point "called_entry" and the resultant entry_object object is associated with the local variable "called_entry_object". The queue of tasks attempting to rendezvous with this entry point of this task is then associated with the identifier "Q". Finally, the name of the task attempting the rendezvous (available through the primitive "current") is used as an index into the task_table to locate the task_object object associated with it which is then referred to through the local variable "calling_task_object". The use of local names to refer to various components of the information structure model occurs in all the communication event definitions and hence will not be discussed again in explaining later event definitions.

The Deterministic_Send event begins by placing the name of the calling task ("current") on the process queue for the desired entry of the specified task. The calling task, which is executing the Deterministic_Send event, is then delayed until the called task is in a position to rendezvous with it. As a result, the calling task must wait until the communication port for the calling task is set to the state "ready_port" by the called task. The calling task resumes execution of the Deterministic_Send event and immediately sets the communication port to the state "busy_port". Since both tasks are now synchronized, the parameters involved in the rendezvous may be passed. Once the calling task has transferred the actual parameters to the called task, the calling task is informed that parameter transmission has taken place. This is done by setting the communication port of the entry within the called task to the state "ready_port". The calling task has little to do now except wait until the statements which make up the body of the rendezvous have been executed by the called task. This is achieved by the calling task waiting until the communication port for the task is set to the "ready_port" state. All that remains is for the calling task to reset the communication port to the "busy_port" state.

5.6.3 Conditional Send

The Conditional_Send event, previously discussed in Section 5.4.3.2, can be described using the alternative SDA approach as shown in Figure 5.33. Recall that a select statement with an else clause is semantically equivalent to a select statement with a timed delay of negative or zero duration. For this reason, the description of the semantics given in Figure 5.33 handles both the else clause and the timed alternative, as the timeout period for the else clause is set to zero.

Figure 5.33 employs the same parameters and local variables as the Deterministic_Send event. As for the Deterministic_Send event, the Conditional_Send event must be instantiated with the appropriate actual parameters representing the actual task

```
Conditional_Send(called_task: task_name_type; called_entry: entry_name_type) =
        using
          called_task_object, calling_task_object: task_object;
          called_entry_object: entry_object;
          Q: queue[task_name_type];
        begin
          called_task_object := task_table(called_task);
          called_entry_object := retrieve_from_table(called_task_object.entry,
                                                     called_entry);
          Q := called_entry_object.process_queue;
          calling_task_object := task_table(current);
          -- Place the current task on the queue for entry "called_entry"
          -- of task "called_task".
          Q := place_on_queue(Q, current);
          do
            -- Wait until the called task can rendezvous, or until timeout expires.
            when ready?(calling_task_object.port) →
              calling_task_object.port := busy_port;
              -- Pass parameters.
              called_entry_object.params := actual_params;
              -- Signal called task that parameter transmission is complete.
              called_entry_object.port := ready_port;
              -- Wait for the rendezvous to complete.
              waituntil ready?(calling_task_object.port) →
                calling_task_object.port := busy_port;
              end waituntil;
            end when;
            when timeout_expired →
              -- Remove current from the entry queue.
              Q := remove_entry(Q, current);
              -- Execute statements in the alternative part of the select statement.
              execute_statements(alternative);
            end when;
          end do;
        end;
```

Figure 5.33. The conditional send event.

and entry point involved. The description of Conditional_Send places the name of the calling task on the process queue of the desired entry within the called task. At this point, one of two conditions may eventually hold: the called task may reach a state where the rendezvous may proceed, or the called task may decide to abandon the rendezvous (represented by the primitive "timeout_expired"); a **do** clause is used to handle this choice. If the called task is able to proceed with the rendezvous, then the first **when** clause within the **do** clause evaluates to true and the statements within it are executed. This is identical to the Deterministic_Send event. If the "timeout_expired" function evaluates to true, then the rendezvous is aborted. This is handled by the second **when** clause within the **do** clause. In this case, the calling task must be removed from the process queue of the called entry point and the statements which comprise the alternative part of the select statement are executed.

5.6.4 Deterministic Receive

The Deterministic_Receive event, described earlier in Section 5.4.3.3, is described in Figure 5.34 in terms of the alternative SDA approach. The event in Figure 5.34 needs only one parameter, the name of the entry_point with which it must deal, as it is invoked by the current task (recall that the current task can be referenced through the primitive "current").

In the case of the Deterministic_Receive event, the called task must wait at the accept statement until some task wishes to rendezvous with that entry point. This is achieved by waiting until the process queue associated with the entry is no longer empty. The rendezvous may now proceed. The called task begins by noting the identity of the calling task in a local variable and then removing it from the process queue. The called task then informs the calling task that the rendezvous may proceed

```
Deterministic_Receive(called_entry: entry_name_type) =
    using
      called_task_object, calling_task_object: task_object;
      called_entry_object := entry_object;
      Q: queue[task_name_type];
    begin
      called_task_object := task_table(current);
      called_entry_object := retrieve_from_table(called_task_object.entry,
                                                  called_entry);
      Q := called_entry_object.process_queue;
      -- Wait until a rendezvous can take place.
      waituntil not queue_empty?(Q) →
        -- Note the calling task.
        calling_task_object := task_table(head_of_queue(Q));
        Q := remove_head(Q);
      end waituntil;
      -- Inform the calling task that parameter transmission may occur.
      calling_task_object.port := ready_port;
      -- Wait until parameter transmission is complete.
      waituntil ready?(called_entry_object.port) →
        called_entry_object.port := busy_port;
      end waituntil;
      -- Execute the statements in the rendezvous.
      execute_statements(rendezvous);
      -- Inform the calling task that the rendezvous is complete.
      calling_task_object.port := ready_port;
    end;
```

Figure 5.34. The deterministic receive event.

and the parameters are passed. The called task then waits until the calling task informs it that the parameters have been transferred. This is achieved by waiting until the communication port for the entry is set to the "ready_port" state. The called task then returns the communication port to the "busy_port" state and executes the statements which comprise the rendezvous; these statements are denoted by the primitive "rendezvous". On completion of the execution of these statements, the called task informs the calling task that the rendezvous is complete by setting the communication port of the called task to the "ready_port" state.

5.6.5 Conditional Receive

The semantics of the Conditional_Receive event, discussed earlier in Section 5.4.3.4, is defined in Figure 5.35. Figure 5.35 shows that the Conditional_Receive event takes many parameters; every entry point that lies within the select statement which results in the Conditional_Receive event is a parameter to the event, as they each have the ability to result in a rendezvous with another task. The Conditional_Receive event starts by making a nondeterministic choice between all of the available options. These options are any of the open entry points (i.e., those entry points whose process queue is nonempty) and the timeout condition, which is represented by an empty process queue for all entry points and a true result from the timeout_expired primitive. If one or more of the entry points has a nonempty process queue, then a nondeterministic choice is made between them and a rendezvous takes place in the same way as that described for Deterministic_Receive. If the timeout condition holds, then the called task need simply execute the statements which make up the alternative part of the select statement and proceed.

5.7 Summary

This chapter has considered the description of language features for parallel programming languages. In keeping with the framework of ATLANTIS described in Chapters 3 and 4, an algebraic framework for the description of data types in a parallel environment has been considered. Mallgren's approach to SDA's allows these parallel features to be described, but fails to integrate well into the ATLANTIS system as language designers would be forced to describe data structures referenced by a single

```
Conditional_Receive(called_entry_1, ..., called_entry_n: entry_name_type) =
    using
      called_task_object, calling_task_object: task_object;
      called_entry_object_1, ..., called_entry_object_n: entry_object;
      Q: queue[task_name_type];
    begin
      called_task_name := task_table(current);
      -- Select one of possibly several open alternatives.
      called_entry_object_1 := retrieve_from_table(called_task_object.entry,
                                                   called_entry_1);
      ...
      called_entry_object_n := retrieve_from_table(called_task_object.entry,
                                                   called_entry_n);
      do
        when not queue_empty?(called_entry_object_1.process_queue) →
          Q := called_entry_object_1.process_queue;
          -- Note the calling task.
          calling_task_object := task_table(head_of_queue(Q));
          Q := remove_head(Q);
          -- Inform the calling task that parameter transmission may commence.
          calling_task_object.port := ready_port;
          -- Wait until parameter transmission is complete.
          waituntil ready?(called_entry_object_1.port) →
            called_entry_object.port := busy_port;
          end waituntil;
          -- Execute the statements in the rendezvous.
          execute_statements(rendezvous);
          -- Inform the calling task that the rendezvous is complete.
          calling_task_object_1.port := ready_port;
        end when;
        when not queue_empty?(called_entry_object_n.process_queue) →
          ...
        end when;
        -- If timeout period expires and no open alternatives.
        when queue_empty?(called_entry_object_1.process_queue)
        and ... and queue_empty?(called_entry_object_n).process_queue)
        and timeout_expired →
          -- Execute statements in the alternative part of the select statement.
          execute_statements(alternative);
        end when;
      end do;
    end;
```

Figure 5.35. The conditional receive event.

process in one manner, and data structures referenced by more than a single process in another.

An alternative approach to the specification of SDA's has been proposed; this approach separates the definition of the data type from the synchronization concerns which govern access to the data a type. This alternative approach lends itself to the automatic generation of an SDA envelope directly from an ADT definition. This permits the ADT description used by ATLANTIS to remain unaltered and hence ATLANTIS can be extended to handle parallel languages without the language designer needing to redesign any of the information structure descriptions. The only modifications visible to the language designer would be the introduction of additional constructs to allow the description of aspects such as the synchronization of processes.

As part of the work on the ATLANTIS system, a number of experiments have been conducted which involve the implementation in Ada of SDA's defined using the alternative technique. The experiments have shown that the technique is free of the majority of difficulties discussed earlier for Mallgren's approach. The principal practical advantage offered by the alternative approach is the fact that an infinite history of previous state transitions is not required.

One difficulty in the above description of the Ada rendezvous occurs if two processes send messages to each other (e.g., in the conditional send) and the messages pass each other enroute. This case is neither handled nor acknowledged by the language definition and no attempt to provide semantics for it is made in the model; to do so would be to present a model which did not accurately reflect the semantics of Ada. It is, however, significant that such an omission comes to light when attempting to formalize the language definition using the alternative approach to SDA's.

Chapter 6

Summary and Conclusions

6.1 Summary

As indicated in Chapter 1, many approaches to the definition of programming languages have been tried, with varying degrees of precision. The syntax of a language is often given in EBNF, or some variant thereof, and this technique is widely accepted and understood. No one technique for the definition of programming language semantics enjoys the same degree of acceptance. The result is that there are many alternative techniques available and no reader of language definitions is familiar with all of them.

The most common method of defining the semantics of a programming language is to employ a natural language description. As is well known, this approach suffers from many problems – ambiguities, omitted details, poorly defined areas and the like. In order to reduce the risk and severity of these difficulties, natural language descriptions have become more precise (at the cost of readability), but even this has not solved the problem completely. It took many person-years of effort to produce relatively precise natural language descriptions for the programming languages Ada and Pascal – but neither is a formal definition. The result is potentially inconsistent interpretations of the definition, and compilers for the same language which may exhibit different behaviours.

In Chapter 2, the algebraic specification technique for the definition of ADT's was discussed. This technique allows the precise, formal definition of a data type and the operations available on it. Such a formalism allows the development of an operational information structure model of the semantics of programming languages, which was introduced in Chapter 3.

The operational semantic model produced by the multi-layer technique described in Chapter 3 is sipler to understand and read than, say, the semantic description of Pascal in terms of attribute grammars given by Kastens *et al.* [68]. An operational model is also capable of describing both the static and dynamic semantics of a language, whilst attribute grammars tend not to perform so well in a description of the dynamic semantics. At present, BSI are attempting to define the semantics of Modula-2 with VDM [13]; however, VDM is itself currently being standardized and will suffer the problems typical of any programming language standard. The axioms used in an axiomatic approach do not suffer any such problem, as they have a firm mathematical foundation.

Some aspects of programming language definition are well explained and understood (such as syntax), whilst others are poorly explained, even amongst formal models of the programming language semantics. For example, very few semantic models have provided adequate descriptions of data control in programming languages. Exceptions include Johnston's Contour Model [67], Smith's Accessing Graph Model [131], Reiss' ACORN project [118], Molinari and Johnson's extension of Reiss' work [96] and the earlier, less formal, versions of the model presented in this thesis [82, 83, 85]. Chapter 3 provides a precise description of the data control aspect of Pascal as an illustration of the operational semantic model developed in that chapter.

The application of the algebraic specification of ADT's to the description of the information structures used in the operational semantic model has led to the development of a layered model that caters for programmers, compiler writers and language designers. Even though each group requires a different depth of understanding of the language, it is now possible to produce one document to satisfy all of these groups, rather than having to write several documents, each aimed at a different group. The algebraic techniques employed gave the degree of formalism necessary to establish a precise base from which to build a formal model.

The approach described in this thesis encourages formality in language design by gathering together activities which have traditionally been distinct: language design, definition and implementation. Generating an implementation from the language definition allows the designer to experiment with and fully explore language design issues with the confidence that the generated implementation precisely matches the formal definition. Language designers are encouraged to produce formal definitions of new languages through the generation of low-cost implementations based directly on their definitions. Such implementations provide language designers with the opportunity to check language definitions to enure that they accurately reflect the intended semantics. Chapter 4 provides a description of the ATLANTIS system. ATLANTIS is based on the operational model developed in Chapter 3 and provides programming language designers and users with an interpretive implementation which faithfully adheres to the language definition.

Since the formal definition is layered, users examining various layers in an ATLANTIS description will all have access to an implementation consistent with the layers concerned. Programmers, normally preferring an informal natural language exposition because such descriptions are easier to read, will be able to answer questions about a language by reading to a depth most appropriate to them. Compiler writers, requiring a more precise definition, are provided with a precise and complete specification based on algebraically-specified abstract data types. Finally, the language designer benefits from using ATLANTIS since an implementation is obtained with little additional effort as a result of designing and defining the language.

Chapter 5 examined the operational model and the ATLANTIS system with respect to the definition of parallel programming languages. This chapter observed that ADT's are inadequate in an environment where multiple concurrent access to an information structure is possible; these difficulties were rectified through the use of SDA's. Chapter 5 also examined Mallgren's approach to SDA's and discovered that such an approach is not compatible with the ATLANTIS system. An alternative approach to SDA's was then discussed; this approach allows the model employing ADT's to remain essentially unchanged except for the introduction of some necessary synchronization mechanisms. The alternative approach provides automatic protection to the ADT's, allowing the language designer to concentrate on the design and definition of a new programming language without being excessively concerned with the protection of data structures in the presence of multiple concurrent processes. The two SDA techniques were illustrated through the definition of aspects of the Ada rendezvous mechanism for inter-task communication.

6.2 Conclusions

The development of the model in Chapter 3 has shown that it is possible to produce a language definition which is formal in nature while simultaneously catering for the disparate needs of various groups of users. This is achieved by layering the underlying operational model so that users requiring a relatively superficial or less detailed understanding can read the outermost layer of the model before resorting to the accompanying natural language narrative, while users such as compiler writers and language designers, who require a detailed and complete understanding, can read each of the layers of the model.

The ATLANTIS system, introduced in Chapter 4, illustrates that a formal language definition can benefit all users, including the language designer, by permitting the generation of a language implementation directly from the definition. The underlying model of ATLANTIS is operational in nature, and the generated interpreter mimics its behaviour precisely. This allows the language designer to experiment with the language definition through experience with the language implementation. If the behaviour of the implementation is determined to be unsatisfactory, or not an accurate reflection of the semantics intended by the language designer, then the language designer may modify the language definition and quickly generate a new implementation. Such feedback to the language designer, which is lacking with some other formalisms, is a major advantage offered by ATLANTIS.

The extension of ATLANTIS from a tool suitable for the definition of a sequential programming language, to a tool suitable for the definition of parallel programming languages has been shown to be relatively straightforward. The changes involved cause minimal disruption to the ATLANTIS system from a user's perspective. This almost transparent extension is possible because the protective SDA envelopes can be generated automatically from the ADT definitions already employed by ATLANTIS. The only disruption to the ATLANTIS system from a user's viewpoint is the introduction of synchronization mechanisms to allow the description of the appropriate features in the programming language concerned.

Another advantage of expanding the operational model through the introduction of SDA's is that it does not introduce any distinctly new style of specification. This is an obvious advantage for those wishing to convert sequential algorithms based on ADT's to a parallel environment. It is not necessary to redefine any ADT specification simply because it will be used within a parallel environment. It is also not necessary to modify the implementation which corresponds to the ADT definition.

6.3 Future Work

Currently, the scanner and the parser are automatically generated from a language definition; the translation of the semantic routines and HLO's to Ada routines is also performed automatically. However, it remains unclear which generation technique is most appropriate for the automatic translation of ADT specifications. In the meantime, ATLANTIS will continue with the hand translation of ADT specifications into Ada.

Although the principal aim of ATLANTIS is not the generation of a production quality compiler generating native code, future development could proceed in this area by employing a code generator generator [35, 36, 47, 48] to produce a compiler. Definition of the necessary instructions for a specific machine architecture could be kept separate from the language definition.

The present ATLANTIS implementation only handles sequential languages. In order to provide the same benefits to the developers of parallel languages, the underlying information structure model needs to be enhanced. Work could proceed in this area through the introduction of a new layer, directly above the ADT layer, to handle several processes trying to simultaneously access the information structures; this layer could then employ the approach to shared data abstractions (SDA's) described in Section 5.5. The introduction of SDA's into ATLANTIS would then provide a language definition technique able to describe both sequential and parallel languages, whilst providing a formal and readable language definition from which an interpreter or compiler prototype can be produced automatically.

The relative ease with which the ATLANTIS system can be extended from sequential languages to parallel languages such as Ada leads to consideration of the classes of languages to which the ATLANTIS approach is amenable. As demonstrated throughout this thesis, but in Chapter 4 in particular, the ATLANTIS system works by devising suitable information structures and expressing the programming language semantics through transformations on these information structures. Furthermore, it is the outermost layer of the model which indicates when these transformations are to take place, by attaching a semantic action to a syntactic element. It clearly follows that any class of language whose semantics can be described in this way can be handled by the ATLANTIS system. A brief consideration of a variety of programming paradigms and programming language features with respect to the approach to language definition adopted by the ATLANTIS system follows.

Sequential block structured languages such as Pascal [17], and the closely related Neptune language used to test the ATLANTIS system, can clearly be described by the technique in this thesis. A parse tree is constructed which represents a program in the programming language defined. The information structures then model activation records and symbol tables, which the generated interpreter manipulates as each semantic action is encountered during a tree walk over the decorated parse tree.

As illustrated in Chapter 5, the approach taken in the ATLANTIS system can be extended to provide support for parallel programming languages. Languages with explicit parallelism (e.g., Ada [141]) can be defined in this way. Other programming languages such as Sisal [32, 89] provide constructs such as parallel for-loops which can be modelled in a similar way to Ada's tasks. Each of these languages provide a concurrency mechanism which is made explicit via some syntactic structure and, hence, provide a convenient syntactic location with which to associate a semantic action to define the necessary semantics through appropriate manipulation of the information structure concerned.

Languages which support implicit parallelism which is detected by the compiler and used to generate efficient code for a particular architecture, such as a supercomputer, typically have no additional semantics associated with parallel constructs. That is to say, the semantics and behaviour of the program written in such a language is unchanged if the construct is parallelized or left sequential. The exploitation of parallelism in these cases is not a language definition issue, but rather a compilation issue. For example, attempts to exploit implicit parallelism in Sisal are handled by the generation of an intermediate form called IF1 [129, 130], and by performing appropriate graph analysis on the generated IF1 code to detect the potential parallelism. Once again, the use of a suitable information structure and its manipulation can be observed; however, in the case of implicit parallelism, the use of the information structure and its manipulation is essentially a compilation technique rather than an aspect of the language definition. The ATLANTIS approach could model such implicit parallelism (possibly with the need for additional primitives to be introduced), but care must be taken by the language designer to separate language issues from compilation and architectural issues. A discussion of these issues is beyond the scope of this thesis.

The object-oriented approach to programming, supported by languages such as Eiffel [92] and C++ [30, 135], typically involves the notions of objects, methods, inheritance and polymorphism. The behaviour of programs in a language such as Eiffel can be modelled by information structures not dissimilar to those used to describe the behaviour of Pascal programs. Clearly, the information structures for Eiffel and Pascal will differ, but there is no reason to believe that object-oriented languages should present a difficult to this approach to language definition. The description of parallel object-oriented languages, such as Eiffel// [19] (a parallel version of Eiffel), will simply employ a mixture of the techniques introduced in Chapters 4 and 5.

Functional languages, such as ML [94], can also be modelled using the approach described in this thesis. Function definitions can be modelled in a similar manner to that used for block structured languages; function applications would be modelled via a variant of the activation record model. A sequential version of Sisal could also be modelled in this way.

Languages which make efficient use of a non von Neumann architecture can also be modelled using the ATLANTIS method. For example, a data-flow language such as Lucid [7, 8] can be modelled in a manner similar to that shown in Wendelborn [153]; Wendelborn provides an operational semantic model of Lucid, making explicit all the implicit parallelism found in this data flow language.

Languages such as Lisp [88, 132], which make no distinction between program and data, cannot easily be modelled by the ATLANTIS system as it currently stands, because there is no mechanism to take some data produced by an executing program, treat it as program text, create an annotated parse tree within the interpreter and then execute it. There is, however, no reason to believe that the ATLANTIS system could not be extended to adequately deal with such languages.

Logic languages, such as Prolog [22, 81] have a relatively simple syntax, yet rich semantics. The difficulty with defining such languages with the approach used by the ATLANTIS system is that the semantics of the inference engine must be encapsulated within the language definition. It is the author's belief that this is possible and that the programming language definition could be written in such a way that the description of the inference engine are separated; this would allow the language designer to easily experiment with different inference semantics for example.

Programming languages which support separate compilation and libraries of routines (e.g., Ada and C [58, 71]) currently present a difficulty to the ATLANTIS system, which assumes that all the necessary context with which to interpret a source program in the language being defined is to be found in either the language definition or the source program. There are no technical difficulties in extending the ATLANTIS approach to cater for separate compilation; library management strategies would need to be explored with such an extension.

Another hurdle not yet addressed by the ATLANTIS system is the description of exception handling mechanisms, such as those found in the languages Ada and PL/I [4]. The principal difficulty with these mechanisms is that it becomes necessary to describe what constitutes an exceptional condition (e.g., divide by zero and memory overflow) and how it is to be detected. Such exceptions are difficult, in most cases, to describe in an architecture-independent manner since they typically relate to limits in the underlying architecture. It is the author's belief that the approach undertaken by the ATLANTIS system could, with substantial modification, be used to describe such exceptional circumstances, but it is likely that some assumptions regarding the underlying architecture will probably need to be made.

One class of language which presents some difficulties for the ATLANTIS approach is where a number of alternative semantic actions are possible and the language designer does not care which choice is made by a compiler writer. For example, a programmer using Ada may not assume that the parameters to a routine are evaluated from left-to-right, right-to-left or even in parallel. All are valid possibilities and each should lead to the same result. If the programmer relies on the fact that the compiler in use employs left-to-right evaluation, for instance, and writes code which exploits this fact, then the programmer has written an erroneous program. This error may, or may not, be detected by the compiler, but may affect the portability of the code written. Ada is defined in this way so as to provide the maximum degree of freedom to the compiler writing team to exploit the architecture being targetted. The ATLANTIS system currently provides no mechanism to express such semantics and, instead, forces the language designer to make a choice, such as the parallel evaluation of parameters, and describe the programming language semantics in terms of this choice. A natural language narrative accompanying the formal description may indicate that alternative implementation strategies are valid. However, the generated interpreter produced by the ATLANTIS system will, of course, follow the implementation scheme described in the underlying operational semantic model. A possible remedy is to permit the language designer to specify a number of alternative strategies and have the interpreter randomly follow one strategy. Furthermore, the ATLANTIS system currently does not demand that all ADT's be consistent and sufficiently complete; this partially serves to address the need to be able to describe "don't care" semantics.

A final deficiency with the current implementation of the ATLANTIS system is the inability to capture the annotated parse tree and use it to generate native code for a particular architecture. This was initially perceived as being beyond the scope of a language definition which should not be bound to any particular architecture; however, the ability to capture the intermediate representation and generate native code would provide a practical and efficient way to, at least partially, produce aspects of a compiler automatically from a language definition. The marriage of a definition of the semantics of a programming language and the definition of the semantics of the native code for a particular architecture, together with an efficient mapping between the two, is an area for continuing research.

To further aid language designers in designing and "debugging" their language design, a graphical interface could be added. The designer would then draw the syntax charts and indicate the locations of the semantic actions on those charts. As a source program in the newly defined language is being interpreted, some indication on the syntax charts could be given to indicate which token is currently under consideration and where in the source program that token was located. The language designer is then in a position to "see" the execution of a sample program and the evaluation of the semantic actions used to define the language. Such tools are not as yet available to language designers; their availability would no doubt improve the quality and accuracy of language definitions and greatly improve the feedback provided to language designers (as discussed in Section 4.11) and further enhance their productivity and the accuracy of the programming language definition.

Appendix A

Predefined ADT's within ATLANTIS

ATLANTIS provides a collection of predefined ADT's. These ADT's are described in this appendix.

A.1 The Boolean Data Type

```
ADT boolean;
   sorts boolean;

   syntax
      true:                                          → boolean;
      false:                                         → boolean;
      and_boolean:       boolean × boolean           → boolean;
      or_boolean:        boolean × boolean           → boolean;
      xor_boolean:       boolean × boolean           → boolean;
      not_boolean:       boolean                     → boolean;
      equal_boolean:     boolean × boolean           → boolean;
      write_boolean:     boolean                     → ;
      read_boolean:                                  → boolean;

   semantics
      -- The operation "true" returns the boolean constant TRUE.
      -- The operation "false" returns the boolean constant FALSE.
      -- The operation "and_boolean" takes two arguments and applies the
      --      boolean operator "and" to them, returning the result.
-- The operation “or_boolean” takes two arguments and applies the
--      boolean operator “or” to them, returning the result.
-- The operation “xor_boolean” takes two arguments and applies the
--      boolean operator “xor” to them, returning the result.
-- The operation “not_boolean” takes a single boolean argument and returns
--      the result after applying the boolean operator “not” to it.
-- The operation “equal_boolean” takes two arguments and returns a
--      boolean value indicating if the arguments were equal.
-- The operation “write_boolean” takes a single boolean value but
--      does not return a result. It produces a side-effect by displaying
--      the boolean value on the screen.
-- The operation “read_boolean” takes no arguments and returns a boolean
--      value read in at the keyboard.
```

A.2 The Integer Data Type

```
ADT integer;
  sorts integer/boolean;

  syntax
    add_integer:            integer × integer    → integer;
    subtract_integer:       integer × integer    → integer;
    multiply_integer:       integer × integer    → integer;
    divide_integer:         integer × integer    → integer ⋃ {error};
    negate_integer:         integer              → integer;
    equal_integer:          integer × integer    → boolean;
    less_than_integer:      integer × integer    → boolean;
    greater_than_integer:   integer × integer    → boolean;
    write_integer:          integer              → ;
    read_integer:                                → integer;

  semantics
    -- The operation “add_integer” adds together two integers and returns the
    --      result.
    -- The operation “subtract_integer” takes the second argument away from
    --      the first and returns the result.
    -- The operation “multiply_integer” multiplies together two integers and
    --      returns the result.
    -- The operation “divide_integer” divides the first argument by the second
    --      giving the result. An error is raised if an attempt to divide
--      by zero is made.
-- The operation “negate_integer” returns the argument after multiplication
--      by −1.
-- The operation “equal_integer” returns a boolean value indicating if the
--      two arguments are equal.
-- The operation “less_than_integer” returns a boolean value indicating if
--      the first argument is less than the second.
-- The operation “greater_than_integer” returns a boolean value indicating if
--      the first argument is greater than the second.
-- The operation “write_integer” takes one argument and produces no result.
--      It has the side-effect of displaying the integer in the screen.
-- The operation “read_integer” reads an integer value from the keyboard
--      and returns the result.
```

A.3 The Floating Point Data Type

```
ADT float;
   sorts float/boolean, integer;

   syntax
      add_float:              float × float        → float;
      subtract_float:         float × float        → float;
      multiply_float:         float × float        → float;
      divide_float:           float × float        → float ∪ {error};
      negate_float:           float                → float;
      equal_float:            float × float        → boolean;
      less_than_float:        float × float        → boolean;
      greater_than_float:     float × float        → boolean;
      write_float:            float                → ;
      read_float:                                  → float;
      integer_to_float:       integer              → float;
      float_to_integer:       float                → integer;

   semantics
      -- The operation “add_float” adds two real numbers together and returns
      --      the result.
      -- The operation “subtract_float” takes the second argument away from
      --      the first and returns the result.
      -- The operation “multiply_float” multiplies two real numbers together
      --      and returns the result.
-- The operation “divide_result” divides the first argument by the second
--      and returns the result. An error is raised if an attempt to divide
--      by zero is made.
-- The operation “negate_float” multiplies its argument by −1 and returns
--      the result.
-- The operation “equal_float” is the test for equality on floating point
--      numbers.
-- The operation “less_than_float” returns a boolean value indicating
--      if the first argument is less than the second.
-- The operation “greater_than_float” returns a boolean value indicating
--      if the second argument is greater than the first.
-- The operation “write_float” takes a real number argument but returns
--      no result. It has the side-effect of displaying the number on the screen.
-- The operation “read_float” reads a real number and returns it.
-- The operation “integer_to_float” takes an integer argument and
--      returns the equivalent floating point number.
-- The operation “float_to_integer” takes a floating point number and
--      rounds it to the nearest integer before returning the result.
```

A.4 The Character Data Type

ADT character;
  sorts character/boolean, integer;

  syntax

equal_char:	character × character	→	boolean;
less_than_char:	character × character	→	boolean;
greater_than_char:	character × character	→	boolean;
write_char:	character	→	;
read_char:		→	character;
char_to_integer:	character	→	integer;
integer_to_char:	integer	→	character;

  semantics

```
-- The operation “equal_char” compares two arguments and returns a
--      boolean result indicating if they are equal.
-- The operation “less_than_char” returns a boolean value indicating
--      if the first argument is less than the second.
-- The operation “greater_than_char” returns a boolean value indicating
--      if the second argument is greater than the second.
-- The operation "write_char" takes a character argument and returns no
--      result. It has the side-effect of displaying the character on the screen.
-- The operation "read_char" returns a character read from the keyboard.
-- The operation "char_to_integer" converts a character to the
--      corresponding integer.
-- The operation "integer_to_char" takes an integer and returns
--      the corresponding character.
```

A.5 The String Data Type

```
ADT string_type;
   sorts string_type/boolean, character, integer, float;

   syntax
      is_empty_string:         string_type                  → boolean;
      concatenate_char:        string_type × character      → string_type;
      concatenate_string:      string_type × string_type    → string_type;
      length_string:           string_type                  → integer;
      equal_string:            string_type × string_type    → boolean;
      less_than_string:        string_type × string_type    → boolean;
      greater_than_string:     string_type × string_type    → boolean;
      write_string:            string_type                  → ;
      read_string:                                          → string_type;
      generate_name:                                        → string_type;
      integer_to_string:       integer                      → string_type;
      string_to_integer:       string_type                  → integer;
      float_to_string:         float                        → string_type;
      string_to_float:         string_type                  → float;
      read_line:                                            → ;
      write_line:                                           → ;
      matching_string:                                      → string_type;
      make_string:             string_type                  → string_type;
      error:                   string_type                  → ;

      define_label:            string_type                  → ;
      define_aux_label:        string_type                  → ;
      return_label:                                         → string_type;
      return_aux_label:                                     → string_type;
```

```
goto_next:          string_type          → ;
goto_prev:          string_type          → ;
```

semantics

```
-- The operation "is_empty_string" returns a boolean value indicating
--      if the string argument is empty.
-- The operation "concatenate_char" concatenates a character to the end
--      of a string and returns the result.
-- The operation "concatenate_string" concatenates the second argument
--      to the end of the first and returns the result.
-- The operation "length_string" returns an integer value indicating the
--      length of the string.
-- The operation "equal_string" returns a boolean value indicating
--      if the two strings are equivalent.
-- The operation "less_than_string" returns a boolean value indicating
--      if the first argument is lexically less than the second.
-- The operation "greater_than_string" returns a boolean value indicating
--      if the first argument is greater than the second.
-- The operation "write_string" takes a string as an argument and returns
--      no result. It has the side-effect of displaying the string on the screen.
-- The operation "read_string" reads a string from the keyboard (terminated
--      by end of line) and returns the result.
-- The operation "generate_name" returns a unique string_type object.
-- The operation "integer_to_string" takes an integer argument and returns
--      the string representation of that number, e.g., 123 ⇒ "123".
-- The operation "string_to_integer" takes a string as an argument and
--      returns the integer which it corresponds to, e.g., "123" ⇒ 123.
-- The operation "float_to_string" takes a real number and converts it
--      to a string, e.g., 12.34 ⇒ "12.34".
-- The operation "string_to_float" takes a string and returns the
--      corresponding floating point number, e.g., "12.34" ⇒ 12.34.
-- The operation "read_line" takes no arguments and returns no results.
--      It has the side-effect of discarding any remaining input on the
--      current input line.
-- The operation "write_line" takes no arguments and returns no result.
--      It has the side-effect of terminating output to the current
--      output line.
-- The operation "matching_string" takes no arguments and returns a
--      (possibly empty) string which corresponds to the string matched
--      by the last token. It operates on the syntax of the language and not
--      the semantics; hence, it is possible to deduce which token will be
--      returned based on the static textual representation of the program text.
-- The operation "error" displays the string as an error message, referring
--      to the line number, and the position on the line, where possible.
```

```
--
-- The next six routines do not really belong in this ADT, but they
--       each have one argument of type string_type.
-- The operation "define_label" specifies the name to be associated with
--      a node in the parse tree.
-- The operation "define_aux_label" specifies the auxiliary label to
--      be associated with the node in the parse tree.
-- The operation "return_label" returns the label associated with the
--      parse node.
-- The operation "return_aux_label" returns the auxiliary label
--      associated with the parse node.
-- The operation "goto_next" changes the active node in the parse tree
--      to the one indicated by the argument. The parse tree is searched
--      forward, but not to a greater depth than what the current level.
-- The operation "goto_prev" performs the same task as "goto_next", but
--      in the opposite direction.
```

A.6 The Location Data Type

```
ADT location;
  sorts location;

  syntax
    current_location:                         → location;
    next_location:                            → location;
    set_location:        location             → ;

  semantics
    -- The operation "current_location" returns a pointer to the current location
    --      in the tree. This is useful when describing the semantics of
    --      procedure call and return.
    -- The operation "next_location" returns a pointer to the location in
    --      the tree which represents the instruction to be executed
    --      after the present instruction.
    -- The operation "set_location" changes the current point is interest
    --      to the specified location in the tree. Again, this is necessary for
    --      the semantics of a procedure call and return.
    -- Together, these operations allow the implementation of procedure and
    -- function call and return by permitting the locations of procedures
-- and functions to be kept in the symbol tables and permitting stacks
-- of locations, etc. Such a facility is sufficiently general to handle
-- all sequential languages and most parallel language features designed
-- to be executed in a sequential environment, e.g., coroutines.
```

Appendix B

ADT implementation

B.1 The Table Package

For each ADT definition in ATLANTIS, there exists an Ada package. This appendix provides an example of the package associated with the ADT of Figure 4.8. The package must be generic in the case of a polymorphic ADT definition. ATLANTIS presumes that the signatures of the operations provided by the package accurately reflect the signatures within the ADT definition; specifically, it is assumed that the operations provided by the package have the same names as those specified in the ADT definition and that the types and ordering of parameters are also preserved, and that the behaviour of the operations corresponds to the axioms. It is also assumed that each package provides a function called "copy" which generates an exact replica, but a distinct object, of any instance of that type.

```
generic
  type index_type is private;
  type element_type is private;

  package table_package is

    type table is private;

    function new_table return table;
      -- Create a new table object.

    function empty_table (tab: in table) return boolean;
      -- Return "true" if the table "tab" is empty; "false" otherwise.

    function member_table (tab: in table;
                           ind: in index_type) return boolean;
      -- Return "true" if the table "tab" contains an element specified
      -- by the index "ind"; returns "false" otherwise.

    function insert_table (tab: in table;
                           ind: in index_type;
                           elem: in element_type) return table;
      -- Insert the specified index and element into the table and return
      -- the newly formed table.

    function remove_table (tab: in table;
                           ind: in index_type) return table;
      -- Remove the index and associated element type from the table.
      -- Raise an exception if the index is not known to the table.

    function alter_table (tab: in table;
                          ind: in index_type;
                          elem: in element_type) return table;
      -- Alter the element value associated with the specified index.  An
      -- exception is raised if the index is not already known to the table.
    function search_table (tab: in table;
                           ind: in index_type) return element_type;
      -- Return the element_type object associated with the index in the
      -- table.  Raise an exception if no such index exists within the table.

    function size_of_table (tab: in table) return integer;
      -- Return the size of the table.

    table_error: exception;

  private
    type table_type;
    type table is access table_type;
    type table_type is
      record
        ind: index_type;
        elem: element_type;
        next: table;
      end record;

  end table_package;



with text_io;
use text_io;

package body table_package is

  function new_table return table is
    -- Create a new table object.

    begin -- new_table
      return null;

    exception
      when others =>
        put_line("Unspecified error occurred in new_table");
        raise;
    end new_table;
```

```
function empty_table (tab: in table) return boolean is
  -- Returns "true" if the table "tab" is empty; returns "false" otherwise.

  begin -- empty_table
    return (tab = null);

  exception
    when others =>
      put_line("Unspecified error occurred in empty_table");
      raise;
  end empty_table;



function member_table (tab: in table;
                       ind: in index_type) return boolean is
  -- Return "true" if the table "tab" contains an element specified by
  -- the index "ind"; return "false" otherwise.

  temp_tab: table := tab;
  found: boolean := false;

  begin -- member_table
    while (not found) and (not empty_table(temp_tab))
    loop
      found := (temp_tab.ind = ind);
      temp_tab := temp_tab.next;
    end loop;
    return found;

  exception
    when others =>
      put_line("Unspecified error occurred in member_table");
      raise;
  end member_table;
```

```
function insert_table (tab: in table;
                       ind: in index_type;
                       elem: in element_type) return table is
  -- Insert the specified index and element into the table and return the
  -- resulting table.

  begin -- insert_table
    return new table_type'(ind, elem, tab);

  exception
    when others =>
      put_line("Unspecified error occurred in insert_table");
      raise;
  end insert_table;



function remove_table (tab: in table;
                       ind: in index_type) return table is
  -- Remove the index and associated element type from the table.
  -- Raise an exception if the index is not known to the table.

  found: boolean := false;
  temp_tab, prev_tab: table;

  begin -- remove_table
    prev_tab := new_table;
    temp_tab := tab;
    while (not found) and (not empty_table(temp_tab))
    loop
      found := temp_tab.ind = ind;
      if not found
      then
        prev_tab := temp_tab;
        temp_tab := temp_tab.next;
      end if;
    end loop;
```

```
      if found
      then
        if not empty_table(prev_tab)
        then
          prev_tab.next := temp_tab.next;
          return tab;
        else
          return temp_tab.next;
        end if;
      else
        raise table_error;
      end if;


    exception
      when table_error =>
        put_line("Attempt to remove an undefined index from the table");
        raise;
      when others =>
        put_line("Unspecified error occurred in remove_table");
        raise;
    end remove_table;



  function alter_table (tab: in table;
                        ind: in index_type;
                        elem: in element_type) return table is
    -- Alter the element value associated with the specified index.
    -- An exception is raised if the index is not already present
    -- in the table.

    found: boolean := false;
    temp_tab: table := tab;
```

```
begin -- alter_table
  while (not found) and (not empty_table(temp_tab))
  loop
    found := (temp_tab.ind = ind);
    if not found
    then
      temp_tab := temp_tab.next;
    end if;
  end loop;


  if found
  then
    temp_tab.elem := elem;
    return tab;
  else
    raise table_error;
  end if;


exception
  when table_error =>
    put_line("Attempt to alter entry not already present in the table");
    raise;
  when others =>
    put_line("Unspecified error occurred in alter_table");
    raise;
end alter_table;




function size_of_table (tab: in table) return integer is
  -- Return the size of the table.

  temp: table := tab;
  result: integer := 0;
  begin -- size_of_table
    while temp /= null
    loop
      result := result + 1;
      temp := temp.next;
    end loop;
    return result;
  end size_of_table;



function search_table (tab: in table;
                       ind: in index_type) return element_type is
  -- Return the element_type object associated with the index in
  -- the table.  Raise an exception if no such index exists within
  -- the table.

  found: boolean := false;
  temp_tab: table := tab;


  begin -- search_table
    while (not found) and (not empty_table(temp_tab))
    loop
      found := (temp_tab.ind = ind);
      if not found
      then
        temp_tab := temp_tab.next;
      end if;
    end loop;


    if found
    then
      return temp_tab.elem;
    else
      raise table_error;
    end if;
  exception
    when table_error =>
      put_line("Attempt to search table for nonexistent index");
      raise;
    when others =>
      put_line("Unspecified error occurred in search_table");
      raise;
  end search_table;

end table_package;
```

Appendix C

ADT's for the Neptune Definition

In order to define the Neptune language, the ADT's described in this appendix were introduced.

C.1 The Stack Data Type

```
ADT stack [element_type];
   sorts stack/element_type, boolean, integer;

   where integer has add_integer: integer × integer → integer;

   syntax
      new_stack:                                           → stack;
      empty_stack:         stack                           → boolean;
      push_stack:          stack × element_type            → stack;
      top_stack:           stack                           → element_type ∪ {error};
      pop_stack:           stack                           → stack ∪ {error};
      size_of_stack:       stack                           → integer;

   semantics
      -- The operation "new_stack" creates and returns a new and empty stack
      --      object.
      -- The operation "empty_stack" returns a boolean value which indicates
      --      whether the argument is an empty stack object.
      -- The operation "push_stack" pushes the second argument onto the stack
--     which is the first argument and returns the result.
-- The operation “top_stack” returns the element_type object which is at the
--     top of the stack which is the first argument. An error is raised if an
--     attempt is made to find the top element of an empty stack.
-- The operation “pop_stack” returns a stack object which is identical to the
--     stack argument with the top element removed.
-- The operation “size_of_stack” returns an integer value which indicates the
--     number of elements stored in the stack argument.
```

C.2 The Table Data Type

```
ADT table [index_type, element_type];
  sorts table/index_type, element_type, boolean, integer;

  where index_type has equal:       index_type × index_type → boolean;
        integer has    add_integer: integer × integer       → integer;

  syntax
    new_table:                                         → table;
    empty_table:      table                            → boolean;
    member_table:     table × index_type               → boolean;
    insert_table:     table × index_type × element_type
                                                       → table;
    remove_table:     table × index_type               → table ⋃ {error};
    alter_table:      table × index_type × element_type
                                                       → table ⋃ {error};
    search_table:     table × index_type               → element_type ⋃ {error};
    size_of_table:    table                            → integer;

  semantics
    -- The operation “new_table” creates and returns a new and empty table
    --     object.
    -- The operation “empty_table” returns a boolean value which indicates
    --     whether the table argument represents an empty table.
    -- The operation “member_table” returns a boolean value which indicates if
    --     the second index_type argument is a member of the first table argument.
    -- The operation “insert_table” returns a table which is a copy of the first
    --     table argument with the index_type/element_type pair inserted. If the
    --     index_type object is already in the table, then the previous value
--      is replaced.
-- The operation "remove_table" returns a table object which is identical to
--      the first table argument with the index_type/element_type pair
--      indexed by the second argument. If the index_type object is not an
--      existing index into table object, then an error is raised.
-- The operation "alter_table" returns a table object which is a copy of the
--      first table argument with the element_type object associated with
--      the second index_type argument replaced by the third argument. An
--      error is raised if the second argument is not an existing index
--      into the first argument.
-- The operation "search_table" returns the element_type object associated
--      with the second index_type argument in the first table argument. An
--      error is raised if the second argument is not an existing index into
--      the first argument.
```

C.3 The Symbol_Table Data Type

```
ADT symbol_table [table];
  sorts symbol_table/table, boolean, string_type;

  syntax
    new_symbol_table:                                              → symbol_table;
    define_father:            symbol_table × symbol_table
                                                                   → symbol_table;
    define_son:               symbol_table × symbol_table
                                                                   → symbol_table;
    define_older_brother:     symbol_table × symbol_table
                                                                   → symbol_table;
    define_younger_brother:   symbol_table × symbol_table
                                                                   → symbol_table;
    define_name:              symbol_table × string_type
                                                                   → symbol_table;
    father:                   symbol_table                         → symbol_table;
    son:                      symbol_table                         → symbol_table;
    older_brother:            symbol_table                         → symbol_table;
    younger_brother:          symbol_table                         → symbol_table;
    define_current_block:     symbol_table × table                 → symbol_table;
    current_block:            symbol_table                         → table ⋃ {error};
    block_name:               symbol_table                         → string_type ⋃ {error};
    empty_symbol_table:       symbol_table                         → boolean;
    info_defined:              symbol_table              → boolean;

semantics
    -- The operation "new_symbol_table" creates a new and empty symbol_table
    --      object.
    -- The operation "define_father" specifies that the second argument is the
    --      father symbol_table of the first argument and returns the result.
    -- The operation "define_son" specifies that the second argument is the son
    --      symbol_table of the first argument and returns the result.
    -- The operation "define_older_brother" specifies that the second argument
    --      is the older_bother symbol_table of the first argument and returns
    --      the result.
    -- The operation "define_younger_brother" specifies that the second argument
    --      is the younger_brother symbol_table of the first argument and returns
    --      the result.
    -- The operation "define_name" attaches the second string argument to the
    --      first symbol_table argument and returns the result.
    -- The operation "father" returns the symbol_table object which is the father
    --      of the argument.
    -- The operation "son" returns the symbol_table object which is the son of
    --      the argument.
    -- The operation "younger_brother" returns the symbol_table object which is
    --      the younger_brother of the argument.
    -- The operation "older_brother" returns the symbol_table object which is
    --       the older_brother of the argument.
    -- The operation "define_current_block" associates the second argument with
    --      the first and returns the result.
    -- The operation "current_block" returns the table object associated with the
    --      symbol_table argument. An error is raised if no table has been
    --      associated with the symbol_table.
    -- The operation "block_name" returns the string associated with the
    --       symbol_table argument. An error is raised if no string is associated
    --      with the symbol_table.
    -- The operation "empty_symbol_table" returns a boolean value indicating
    --      if any information (of type table) has been associated with the
    --      symbol_table, or any of father, son, older_brother or younger_brother
    --       are defined.
    -- The operation "info_defined" returns a boolean value indicating if an
    --      object of type table is associated with the symbol_table.
```

C.4 The Block_Info Data Type

```
ADT block_info [attributes];
   sorts block_info/string_type, attributes, integer, boolean;

   where string_type has equal_string:  string_type × string_type → boolean;
         integer has     equal_integer: integer × integer         → boolean;
         integer has     add_integer:   integer × integer         → integer;

   syntax
      new_block:                                         → block_info;
      empty_block:            block_info                 → boolean;
      add_to_block:           block_info × string_type × attributes
                                                         → block_info;
      insert_into_block:      block_info × string_type × attributes
                                                         → block_info;
      delete_from_block:      block_info × string_type   → block_info ∪ {error};
      member_of_block:        block_info × string_type   → boolean;
      associated_attributes:  block_info × string_type   → attributes;
      get_object_i:           block_info × integer       → string_type ∪ {error};
      get_attributes_i:       block_info × integer       → attributes ∪ {error};
      size_of_block:          block_info                 → integer;
      alter_block:            block_info × string_type × attributes
                                                         → block_info ∪ {error};
      alter_member_i:         block_info × integer × attributes
                                                         → block_info ∪ {error};

   semantics
      -- The operation "new_block" creates a new and empty block_info object
      --     and returns it.
      -- The operation "empty_block" returns a boolean value indicating whether
      --     the block_info argument represents an empty block_info object.
      -- The operation "add_to_block" associates the third attributes argument
      --     with the second string_type argument in the first argument and returns
      --     the resulting block_info object.
      -- The operation "insert_into_block" has the same effect as add_to_block
      --     if the second argument is not already part of the block_info object;
      --     otherwise, the attributes associated with the string_type argument
      --     are updated.
      -- The operation "delete_from_block" removes the second argument, and its
      --     associated attributes, from the first block_info argument and returns
      --     the result. An error is raised if the second argument is not in the
--      block_info object.
-- The operation "member_of_block" returns a boolean value indicating if
--      the second string_type argument is found within the first block_info
--      argument.
-- The operation "associated_attributes" returns the attributes associated
--      with the second string_type argument in the first block_info argument.
--      An error is raised if the second argument is not in the block_info
--      object.
-- The operation "get_object_i" returns the string_type object associated
--      with the ith element of the first block_info argument. An error
--      is raised if the ith element does not exist.
-- The operation "get_attributes_i" returns the attributes object associated
--      with the ith element of the first block_info argument. An error
--      is raised if the ith element does not exist.
-- The operation "size_of_block" returns an integer value indicating the
--      number of elements in the block_info object.
-- The operation "alter_block" associates the third attributes argument
--      replacing any existing attributes) with the second string_type
--      argument in the first block_info argument and returns the result.
--      An error is raised if the second string_type argument is not
--      found in the block_info object.
-- The operation "alter_member_i" alters the attributes associated with the
--      ith element of the block_info object returning the resultant block_info
--      object. An error is raised if the ith element does not exist.
```

C.5 The Declaration Data Type

ADT declaration;
  sorts declaration/boolean;

  syntax

local:		→	declaration;
nonlocal:		→	declaration;
equal_declaration:	declaration × declaration	→	boolean;

  semantics

```
-- The operation "local" returns a declaration object with the value local.
-- The operation "nonlocal" returns a declaration object with the value
--      nonlocal.
-- The operation "equal_declaration" compares two declaration objects, and
--      returns a boolean result whose value is true if they are the same and
--      false otherwise.
```

C.6 The Initial_Value Data Type

```
ADT initial_value;
  sorts initial_value/boolean;

  syntax
    initialized:                                        → initial_value;
    not_initialized:                                    → initial_value;
    equal_initial:      initial_value × initial_value → boolean;

  semantics
    -- The operation "initialized" returns an initial_value object with the
    --      value initialized.
    -- The operation not_initialized returns an initial_value object with the value
    --      not_initialized.
    -- The operation "equal_initial" compares two initial_value objects, and
    --      returns a boolean value whose value is true if they are the same
    --      and false otherwise.
```

C.7 The Kind Data Type

```
ADT kind;
  sorts kind/boolean;

  syntax
    void_kind:                                          → kind;
    variable_kind:                                      → kind;
    procedure_kind:                                     → kind;
    function_kind:                                      → kind;
    parameter_kind:                                     → kind;
    equal:              kind × kind                     → boolean;
```

```
   semantics
      -- The operation "void_kind" returns a kind object with the value
      --      void_kind.
      -- The operation "variable_kind" returns a kind object with the value
      --      variable_kind.
      -- The operation "procedure_kind" returns a kind object with the value
      --      procedure_kind.
      -- The operation "function_kind" returns a kind object with the value
      --      function_kind.
      -- The operation "parameter_kind" returns a kind object with the value
      --      parameter_kind.
      -- The operation "equal" compares two kind objects, and returns true if
      --      they are the same and false otherwise.
```

C.8 The Operation Data Type

```
ADT operation;
   sorts operation/boolean;

   syntax
      equal_op:                                              → operation;
      and_op:                                                → operation;
      or_op:                                                 → operation;
      not_equal_op:                                          → operation;
      less_than_op:                                          → operation;
      greater_than_op:                                       → operation;
      less_or_equal_op:                                      → operation;
      greater_or_equal_op:                                   → operation;
      plus_op:                                               → operation;
      minus_op:                                              → operation;
      mult_op:                                               → operation;
      divide_op:                                             → operation;
      equal:                 operation × operation           → boolean;

   semantics
      -- The operation "equal_op" returns an operation object whose value is
      --      equal_op.
      -- The operation "and_op" returns an operation object whose value is
--      and_op.
-- The operation “or_op” returns an operation object whose value is or_op.
-- The operation “not_equal_op” returns an operation object whose value is
--      not_equal_op.
-- The operation “less_than_op” returns an operation object whose value is
--      less_than_op.
-- The operation “greater_than_op” returns an operation object whose
--      value is greater_than_op.
-- The operation “less_or_equal_op” returns an operation object whose
--      value is less_or_equal_op.
-- The operation “greater_or_equal_op” returns an operation object
--      whose value is greater_or_equal_op.
-- The operation “plus_op” returns an operation object whose value is
--      plus_op.
-- The operation “minus_op” returns an operation object whose value
--      is minus_op.
-- The operation “mult_op” returns an operation object whose value is
--      mult_op.
-- The operation “divide_op” returns an operation object whose value
--      is divide_op.
-- The operation “equal” returns a boolean value indicting whether the
--      two operation arguments are the same.
```

C.9 The Basic_Types Data Type

```
ADT basic_types;
  sorts basic_types/boolean;

  syntax
    void_ty:                                          → basic_types;
    integer_ty:                                       → basic_types;
    float_ty:                                         → basic_types;
    boolean_ty:                                       → basic_types;
    string_ty:                                        → basic_types;
    equal:              basic_types × basic_types     → boolean;

  semantics
    -- The operation “void_ty” returns a basic_types object with the value
    --      void_ty.
-- The operation "integer_ty" returns a basic_types object with the value
--      integer_ty.
-- The operation "float_ty" returns a basic_types object with the value
--      float_ty.
-- The operation "boolean_ty" returns a basic_types object with the value
--      boolean_ty.
-- The operation "string_ty" returns a basic_types object with the value
--      string_ty.
-- The operation "equal" returns a boolean value indicating if two
--      basic_types objects are the same.
```

C.10 The Attributes Data Type

```
ADT attributes;
  sorts attributes/string_type, kind, basic_types, declaration,
        initial_value, integer, float, boolean, location;

  syntax
    new_attributes:                                   → attributes;
    insert_name:            attributes × string_type  → attributes;
    insert_kind:            attributes × kind         → attributes;
    insert_type:            attributes × basic_types  → attributes;
    insert_declaration:     attributes × declaration  → attributes;
    insert_initial:         attributes × initial_value → attributes;
    insert_integer_value:   attributes × integer      → attributes;
    insert_float_value:     attributes × float        → attributes;
    insert_string_value:    attributes × string_type  → attributes;
    insert_boolean_value:   attributes × boolean      → attributes;
    insert_start_location:  attributes × location     → attributes;
    return_name:            attributes                → string_type ⋃ {error};
    return_kind:            attributes                → kind ⋃ {error};
    return_type:            attributes                → basic_types ⋃ {error};
    return_declaration:     attributes                → declaration ⋃ {error};
    return_initial:         attributes                → initial_value ⋃ {error};
    return_integer_value:   attributes                → integer ⋃ {error};
    return_float_value:     attributes                → float ⋃ {error};
    return_string_value:    attributes                → string_type ⋃ {error};
    return_boolean_value:   attributes                → boolean ⋃ {error};
    return_start_location:  attributes                → location ⋃ {error};
```

semantics

```
-- The operation "new_attributes" creates and returns a new and empty
--      attributes object.
-- The operation "insert_name" inserts the second string_type argument
--      into the first attributes argument and returns the resulting attributes
--      object.
-- The operation "insert_kind" inserts the second kind argument into the
--      first attributes argument and returns the resulting attributes object.
-- The operation "insert_type" inserts the second basic_types argument into
--      the first attributes argument and returns the resulting attributes
--      object.
-- The operation "insert_declaration" inserts the second declaration argument
--      into the first attributes argument and returns the resulting attributes
--      object.
-- The operation "insert_initial" inserts the second initial_value argument
--      into the first attributes argument and returns the resulting attributes
--      object.
-- The operation "insert_integer_value" inserts the second integer argument
--      into the first attributes argument and returns the resulting attributes
--      object.
-- The operation "insert_float_value" inserts the second floating point
--      argument into the first attributes argument and returns the resulting
--      attributes object.
-- The operation "insert_string_value" inserts the second string_type argument
--      into the first attributes argument and returns the resulting attributes
--      object.
-- The operation "insert_boolean_value" inserts the second boolean argument
--      into the first attributes argument and returns the resulting attributes
--      object.
-- The operation "insert_start_location" inserts the second location argument
--      into the first attributes argument and returns the resulting attributes
--      object.
-- The operation "return_name" returns the string_type value associated
--      with the attributes object through the define_name operation. An
--      error is raised if no string_type object is associated with the attributes
--      object.
-- The operation "return_kind" returns the kind value associated with the
--      attributes object. An error is raised if no kind object is associated
--      with the attributes object.
-- The operation "return_type" returns the basic_types value associated with
--      the attributes object. An error is raised if no basic_types object
--      is associated with the attributes object.
-- The operation "return_declaration" returns the declaration value associated
--      with the attributes object. An error is raised if no declaration object
--      is associated with the attributes object.
-- The operation “return_initial” returns the initial_value value associated
--      with the attributes object. An error is raised if no initial_value
--      object is associated with the attributes object.
-- The operation “return_integer_value” returns the integer value associated
--      with the attributes object. An error is raised if no integer object
--      is associated with the attributes object.
-- The operation “return_float_value” returns the floating point value
--      associated with the attributes object. An error is raised if no floating
--      point object is associated with the attributes object.
-- The operation “return_string_value” returns the string_type value
--      associated with the attributes object through the insert_string_value
--      operation. An error is raised if no string_type object is associated
--      with the attributes object.
-- The operation “return_boolean_value” returns the boolean value associated
--      with the attributes object. An error is raised if no boolean object
--      is associated with the attributes object.
-- The operation “return_start_location” returns the location value associated
--      with the attributes object. An error is raised if no location object
--      is associated with the attributes object.
```

Appendix D

HLO's from the Neptune Definition

This appendix outlines the HLO's used in the Neptune definition.

D.1 Procedure Initialize

```
procedure initialize is ...
   -- This HLO initializes the environment. It creates the necessary
   -- information structures and provides them with an initial value.
```

D.2 Procedure Reset_Environment

```
procedure reset_environment is ...
   -- Reset the environment ready for a new pass.
```

D.3 Procedure Check_Block

```
procedure check_block_name (str: string_type) is ...
   -- Check that the name used at the end of the block is the same as that
   -- used when the block was declared.
```

D.4 Procedure Create_Block

```
procedure create_block (str: string_type; in_function: boolean) is ...
   -- Establish data structures necessary for a new block. Also note
   -- if the block entered is a procedure or a function, as this affects
   -- whether or not a return statement is valid.
```

D.5 Procedure Enter_Block

```
procedure enter_block (str: string_type) is ...
   -- Locate the block with the specified name. It must be a sibling of
   -- the current symbol table.
```

D.6 Procedure Exit_Block

```
procedure exit_block is ...
   -- Leave a block. Reset any relevant data structures; for example, set
   -- the current symbol table back to that for the parent block.
```

D.7 Procedure Leave_Block

```
procedure leave_block is ...
   -- Return to parent block.
```

D.8 Procedure Note_Identifier

```
procedure note_identifier (str: string_type) is ...
   -- Record the identifier by pushing it onto the identifier stack.
```

D.9 Function Recall_Identifier

function recall_identifier **return** string_type **is** ...
  -- Recall the identifier at the top of the identifier stack.

D.10 Procedure Note_Previous_Arguments

procedure note_previous_arguments **is** ...
  -- Note the current arguments in the argument stack, so that the latter
  -- can be restored later.

D.11 Procedure Recall_Previous_Arguments

procedure recall_previous_arguments **is** ...
  -- Restore the arguments to the state they were in prior to the routine call.

D.12 Procedure Record_Type

procedure record_type (ty: basic_types) **is** ...
  -- Record the type on the type stack so that it can be used later.

D.13 Procedure Record_Op

procedure record_op (op: operation) **is** ...
  -- Record an operation in the operation stack.

D.14 Procedure Record_Integer

procedure record_integer(int: integer) **is** ...
  -- Push an integer onto the integer stack.

D.15 Procedure Record_Real

```
procedure record_real(real: float) is ...
   -- Push a real number onto the real stack.
```

D.16 Procedure Record_Boolean

```
procedure record_boolean (bool: boolean) is ...
   -- Push a boolean value onto the boolean stack.
```

D.17 Procedure Record_String

```
procedure record_string(str: string_type) is ...
   -- Push a string onto the top of the string stack.
```

D.18 Procedure Reverse_Ident_Stack

```
procedure reverse_ident_stack is ...
   -- Reverse the contents of the identifier stack. This is necessary in
   -- order to insert identifiers into the symbol table in the order in
   -- which they were declared.
```

D.19 Procedure Add_To_Symbol_Table

```
procedure add_to_symbol_table (kind_of_object: kind) is ...
   -- Add all the objects in identifier stack to the symbol table represented
   -- by the current block. The identifiers should be inserted in the order
   -- in which they were declared; this will necessitate inverting the stack.
   -- The type associated with the objects is found on the type stack.
```

D.20 Procedure Add_Procedure_To_Symbol_Table

```
procedure add_procedure_to_symbol_table (str: string_type) is ...
   -- Add a string to the current symbol table as the name of a procedure.
```

D.21 Procedure Add_Function_To_Symbol_Table

```
procedure add_function_to_symbol_table (str: string_type) is ...
   -- Add a function to the appropriate symbol table – the latter will be
   -- the parent of the current symbol table.
```

D.22 Procedure Ensure_Within_Function

```
procedure ensure_within_function is ...
   -- Ensure that the current context is within a function. Write out an
   -- error message if this is not so.
```

D.23 Procedure Ensure_Within_Procedure

```
procedure ensure_within_procedure is ...
   -- Ensure that the current context is within a procedure. Write out an
   -- error message if this is not so.
```

D.24 Procedure Ensure_Within_Loop

```
procedure ensure_within_loop is ...
   -- Ensure that the current context is within a loop. Write out an
   -- error message if this is not so.
```

D.25 Procedure Start_Loop

```
procedure start_loop is ...
   -- Note that a loop has begun and, as a result, exit statements are valid.
```

D.26 Procedure Finish_Loop

```
procedure finish_loop is ...
   -- Loop processing has finished. Note that exit statements may no
   -- longer be valid.
```

D.27 Procedure Inherit_Via_Scope_Rules

```
procedure inherit_via_scope_rules (decl: declaration) is ...
   -- Inherit from the parent block any identifiers which are not already
   -- present in the current block, provided that they are defined as being
   -- local or nonlocal as specified by the parameter "decl".
```

D.28 Procedure Record_Start_Of_Block

```
procedure record_start_of_block (str: string_type) is ...
   -- Record the starting location of the block whose name is "str".
```

D.29 Procedure Return_To_Caller

```
procedure return_to_caller is ...
   -- Return to the calling block and continue execution there. This is
   -- achieved by resetting several environment variables and resetting the
   -- current location within the generated parse tree.
```

D.30 Function Reverse_Type_Stack

function reverse_type_stack(stk: type_stack) **return** type_stack **is** ...
    -- Reverse the contents of the argument stack.

D.31 Procedure Transfer_To_Argument_List

procedure transfer_to_argument_list **is** ...
    -- Take the value (type) from the top of the type stack and put it on
    -- the argument stack.

D.32 Procedure Check_Call_To_Procedure

procedure check_call_to_procedure (str: string_type) **is** ...
    -- Check that a string represents a procedure to the current block.

D.33 Procedure Check_Call_To_Function

procedure check_call_to_function (str: string_type) **is** ...
    -- Check that a string represents a function to the current block.

D.34 Procedure Check_Call_To_Variable

procedure check_call_to_variable (str: string_type) **is** ...
    -- Check that a string represents a variable or parameterless function
    -- to the current block.

D.35 Procedure Check_Boolean_Type

procedure check_boolean_type **is** ...
    -- Check that the top of the type stack represents a boolean value.

D.36 Procedure Check_Numeric_Type

```
procedure check_numeric_type is ...
   -- Check that the top of the type stack represents a number.
```

D.37 Procedure Negate_Expression

```
procedure negate_expression is ...
   -- Check that the top of the stack is an integer or a real number, and then
   -- negate it. Otherwise, issue an error message.
```

D.38 Procedure Not_Expression

```
procedure not_expression is ...
   -- Check that the top of the stack represents a boolean value and then
   -- negate it. Otherwise, issue an error message.
```

D.39 Procedure Determine_Exit_Statement

```
procedure determine_exit_statement (str: string_type) is ...
   -- Determine if the exit statement indicates that the current loop should
   -- be left.
```

D.40 Procedure Determine_Branch

```
procedure determine_branch (str: string_type) is ...
   -- Determine which branch to take in a conditional statement.
```

D.41 Function Compatible_Types

```
function compatible_types(ty_1, ty_2: basic_types) return boolean is ...
   -- Return a boolean value indicating whether the two types are compatible.
```

D.42 Procedure Equality_Test

```
procedure equality_test is ...
   -- Perform a test for equality and leave the boolean result on the
   -- appropriate stack.
```

D.43 Procedure Inequality_Test

```
procedure inequality_test is ...
   -- Perform a test for inequality and leave the boolean result on
   -- the appropriate stack.
```

D.44 Procedure And_Operation

```
procedure and_operation is ...
   -- Perform an "and" operation and leave the boolean result on the
   -- appropriate stack.
```

D.45 Procedure Or_Operation

```
procedure or_operation is ...
   -- Perform an "or" operation and leave the boolean result on the
   -- appropriate stack.
```

D.46 Procedure Less_Than_Operation

```
procedure less_than_operation is ...
   -- Perform a "<" comparison and leave the boolean result on the
   -- appropriate stack.
```

D.47 Procedure Greater_Than_Operation

```
procedure greater_than_operation is ...
   -- Perform a ">" comparison and leave the boolean result on the
   -- appropriate stack.
```

D.48 Procedure Less_Or_Equal_Operation

```
procedure less_or_equal_operation is ...
   -- Perform a "≤" comparison and leave the boolean result on the
   -- appropriate stack.
```

D.49 Procedure Greater_Or_Equal_Operation

```
procedure greater_or_equal_operation is ...
   -- Perform a "≥" comparison and leave the boolean result on the
   -- appropriate stack.
```

D.50 Procedure Plus_Operation

```
procedure plus_operation is ...
   -- Perform an addition operation and leave the result on the appropriate stack.
```

D.51 Procedure Minus_Operation

```
procedure minus_operation is ...
   -- Perform a subtraction operation and leave the result on the appropriate stack.
```

D.52 Procedure Mult_Operation

```
procedure mult_operation is ...
   -- Perform a multiplication operation and leave the result on the appropriate stack.
```

D.53 Procedure Divide_Operation

```
procedure divide_operation is ...
   -- Perform a division operation and return the result on the appropriate stack.
```

D.54 Procedure Evaluate_Expression

```
procedure evaluate_expression is ...
   -- Evaluate the expression and leave the result on the appropriate stack.
```

D.55 Procedure Perform_Output

```
procedure perform_output is ...
   -- Write a value to standard output, which is normally the screen.
```

D.56 Procedure Perform_Input

```
procedure perform_input(str: string_type) is ...
   -- Read a value from the keyboard and store it in the appropriate location.
```

D.57 Procedure Perform_Assignment

```
procedure perform_assignment (str: string_type) is ...
   -- Perform assignment to the variable whose name is given by the specified
   -- string. The value is stored on the stack. Type checking also takes place.
```

D.58 Procedure Invoke_Block

```
procedure invoke_block (str: string_type) is ...
   -- Invoke the block specified by "str".
```

D.59 Procedure Invoke_Block_Or_Variable

```
procedure invoke_block_or_variable (str: string_type) is ...
   -- If "str" is the name of a procedure or function, then invoke
   -- it; otherwise, place the value of the variable, and its type, on the
   -- appropriate stacks.
```

Bibliography

[1] K. Abrahamson. Modal logic of concurrent nondeterministic programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, pages 21–33, Berlin, 1979. Springer-Verlag. Volume 70 of *Lecture Notes in Computer Science.*

[2] L. Allison. *A Practical Introduction to Denotational Semantics.* Cambridge University Press, Cambridge, 1986.

[3] P. America, J.W. de Bakker, J.N. Kok and J.J.M.M. Rutten. A denotational semantics of a parallel object-oriented language. Technical Report CS-R8626, Centrum voor Wiskunde en Informatica, Computer Science/Department of Software Technology, Amsterdam, Holland, August 1986.

[4] ANSI. *American National Standard Programming Language PL/I (ANS X3.53-1976).* American National Standards Institute, New York, 1976.

[5] K.R. Apt, N. Francez and W.P. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, Volume 2, Number 3, pages 359–385, July 1980.

[6] M.A. Arbib and E.G. Manes. Parameterized data types do not need highly constrained parameters. *Information and Control*, Volume 52, Number 2, pages 139–158, February 1982.

[7] E.A. Ashcroft and W.W. Wadge. Lucid, a non-procedural language with iteration. *Communications of the ACM*, Volume 20, Number 7, pages 519–526, July 1977.

[8] E.A. Ashcroft and W.W. Wadge. Structured Lucid. Technical Report CS-79-21, Department of Computer Science, The University of Waterloo, Waterloo, Canada, June 1979.

[9] J.G.P. Barnes. *Programming in Ada.* Addison–Wesley, Wokingham, England, third edition, 1989.

[10] M. Bidoit and C. Choppy. ASSPEGIQUE: An integrated environment for algebraic specification. In H. Ehrig, C. Floyd, M. Nivat and J. Thatcher (editors), *Formal Methods and Software Development*, pages 246–260, Berlin, 1984. Springer-Verlag. Volume 186 of *Lecture Notes in Computer Science.*

[11] M. Bidoit, C. Choppy and F. Voisin. The ASSPEGIQUE specification environment – motivations and design. In H.-J. Kreowski (editor), *Recent Trends in Data Type Specification. Third Workshop on Theory and Applications of Abstract Data Types – Selected Papers*, pages 54–74. Springer-Verlag, 1985. Volume 116 of *Informatik-Fachberichte.*

[12] P. Bird. An implementation of a code generator specification language for table driven code generators. *ACM SIGPLAN Notices*, Volume 17, Number 6, pages

44–55, June 1982. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Boston, Massachusetts, 23–25 June, 1982.

[13] D. Bjørner and C.B. Jones (Editors). *The Vienna Development Method: The Meta-Language*, Volume 61 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1978.

[14] D.C.C. Bover, K.J. Maciunas and M.J. Oudshoorn. *Ada. A First Course in Programming and Software Engineering.* Addison–Wesley, Sydney, 1991.

[15] BSI. Minutes of the first meeting of ISO/TC97/SC22/WG13. Held at the University of Nottingham on 1^{st}–3^{rd} April, 1987.

[16] BSI. Working draft of standard Pascal by the BSI DPS/13/14 working group. *Pascal News*, Volume 14, January 1979.

[17] BSI. *Specification For Computer Programming Language Pascal.* British Standards Institution, London, 1982. Publication BS 6192:1982.

[18] R.M. Burstall and J.A. Goguen. The semantics of Clear, a specification language. Technical Report CSR-65-80, Department of Computer Science, University of Edinburgh, February 1980.

[19] D. Caromel. *Programmation Parallèle Asynchrone et Impérative: Etudes et Propositions.* Ph.D. thesis, L'Universite de Nancy I, Nancy, France, 1991.

[20] J.C. Cherniavsky and S.N. Kamin. A complete and consistent Hoare axiomatics for a simple programming language. *Journal of the ACM*, Volume 26, Number 1, pages 119–128, January 1979.

[21] C. Choppy. ASSPEGIQUE user's manual. Rapport de Recherche 452, L.R.I., October 1988.

[22] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[23] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *ACM SIGPLAN Notices*, Volume 24, Number 10, pages 433–444, October 1989. Proceedings of the OOPSLA'89 Conference. Object-Oriented Programming: Systems, Languages and Applications, New Orleans, Louisiana, 1–6 October, 1989.

[24] D. Cooper. *Standard Pascal User Reference Manual.* W.W. Norton & Company, New York, 1983.

[25] A.J. Demers and J.E. Donahue. Data types, parameters and type checking. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 12–23, Las Vagas, Nevada, January 1980.

[26] C.A.P. Denbaum. *A Demand Driven, Coroutine-Based Implementation of a Nonprocedural Language.* Ph.D. thesis, Department of Computer Science, The University of Iowa, Iowa City, Iowa, May 1983. Available as Technical Report 83-01.

[27] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys (editor), *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[28] A. Diller. *Z. An Introduction to Formal Methods.* John Wiley and Sons, Chichester, England, 1990.

[29] R. Duke, D. Johnston and G.A. Rose. Specifying the static semantics of block structured langauges. *Australian Computer Journal*, Volume 19, Number 2, pages 99–104, May 1987.

[30] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison–Wesley, Reading, Massachusetts, 1990.

[31] R. Farrow. LINGUIST-86. Yet another translator writing system based on attribute grammars. *ACM SIGPLAN Notices*, Volume 17, Number 6, pages 160–171, June 1982. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Boston, Massachusetts, 23–25 June, 1982.

[32] J.T. Feo, D.C. Cann and R.R. Oldehoeft. A report on the Sisal language project. *Jounral of Parallel and Distributed Computing*, Volume 10, Number 12, pages 349–366, December 1990.

[33] D.H. Freidel. *Modelling Communication and Synchronization in Parallel Programming Languages.* Ph.D. thesis, Department of Computer Science, The University of Iowa, Iowa City, Iowa, May 1984. Available as Technical Report 84-01.

[34] D.H. Freidel, C.D. Marlin and M.J. Oudshoorn. Modelling communication in Ada with shared data abstractions. Technical Report 88-06, Department of Computer Science, The University of Adelaide, Adelaide, South Australia, December 1988. (Revised September 1989).

[35] M. Ganapathi and C.N. Fischer. Description–driven code generation using attribute grammars. In *Conference Record of the Ninth Annual ACM Symposium*

on Principles of Programming Languages, pages 108–119, Albuquerque, New Mexico, January 1982.

[36] M. Ganapathi and C.N. Fischer. Affix grammar driven code generation. *ACM Transaction an Programming Languages and Systems*, Volume 7, Number 4, pages 560–599, October 1985.

[37] H. Ganzinger. Denotational semantics for languages with modules. In D. Bjørner (editor), *Formal Description of Programming Concepts*, pages 3–23. North–Holland, Amsterdam, 1982.

[38] H. Ganzinger. Parameterized specifications: Parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems*, Volume 5, Number 3, pages 318–345, July 1983.

[39] N. Gehani. *Ada. An Advanced Introduction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[40] R.S. Glanville and S. Graham. A new method for compiler code generation. In *Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 231–239, Tucson, Arizona, January 1978.

[41] J.A. Goguen. Correctness and equivalence of data types. In *Mathematical Systems Theory, Proceedings of the Initial Symposium*, pages 352–358. Springer-Verlag, Berlin, 1975. Volume 131 of *Lecture Notes in Economics and Mathematical Systems.*

[42] J.A. Goguen. Abstract errors for abstract data types. In E.J. Neuhold (editor), *Formal Descriptions of Programming Concepts*, pages 491–526. North-Holland, Amsterdam, 1978.

[43] J.A. Goguen, J.W. Thatcher and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R.T. Yeh (editor), *Current Trends in Programming Methodology*, Volume 4, Chapter 5, pages 80–149. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[44] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, Volume 24, Number 1, pages 68–95, 1977.

[45] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 1988.

[46] M.J.C. Gordon. *The Denotational Description of Programming Languages. An Introduction.* Springer-Verlag, New York, New York, 1979.

[47] S.L. Graham. Table–driven code generation. *IEEE Computer*, Volume 13, Number 8, pages 25–34, August 1980.

[48] S.L. Graham, R.R. Henry and R.A. Schulman. An experiment in table driven code generators. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, pages 32–43, Boston, Massachusetts, June 1982. (Also in *ACM SIGPLAN Notices*, Volume 17, Number 6, June 1982).

[49] I. Guessarian and J. Meseguer. Axiomatisation of "IF...THEN...ELSE" revisited. Technical Report 84-18, Laboratoire Informatique Theorique et Programmation, Université P. et M. Curie, Paris, April 1984.

[50] J.V. Guttag. The specification and application to programming of abstract data types. Technical Report CSRG-59, Department of Electrical Engineering and the Department of Computer Science, The University of Toronto, Toronto, Ontario, September 1975.

[51] J.V. Guttag. Notes on type abstraction (Version 2). *IEEE Transactions on Software Engineering*, Volume SE–6, pages 13–23, January 1980.

[52] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, Volume 10, Number 1, pages 27–52, 1978.

[53] J.V. Guttag, J.J. Horning and J.M. Wing. Some notes on putting formal specifications to productive use. Technical Report CSL–82–3, Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California, June 1982.

[54] J.V. Guttag, J.J. Horning and J.M. Wing. The Larch family of specification languages. *IEEE Software*, Volume 2, Number 5, pages 24–36, September 1985.

[55] J.V. Guttag, E. Horowitz and D.R. Musser. The design of data type specifications. In R.T. Yeh (editor), *Current Trends in Programming Methodology*, Volume 4, Chapter 4, pages 60–79. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[56] B. Hailpern and S. Owicki. Modular verification of concurrent programs. In *Record of the Ninth ACM Symposium on Principles of Programming Languages*, pages 322–336, Albuquerque, New Maxico, January 1982.

[57] J.Y. Halpern. A good Hoare axiom system for an Algol-like language. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 262–271, Salt Lake City, Utah, January 1984.

[58] S.P. Harbison and G.L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1987.

[59] I. Hayes (editor). *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, Engelwood Cliffs, New Jersey, 1987.

[60] C.A.R. Hoare. An axiomatic definition of the programming language Pascal. *Acta Informatica*, Volume 2, Number 4, pages 335–355, 1973.

[61] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, Volume 17, Number 10, pages 549–557, October 1974.

[62] Honeywell, Inc. *Formal Definition of Ada.* Interim Draft, Systems and Research Center, Honeywell Inc., Minneapolis, Minnesota, October 1979.

[63] ISO. *Second DP 7185 – Specification for the Computer Programming Language Pascal.* International Organization for Standardization (ISO), December 1980.

[64] ISO. *First DP 7185 – Specification for the Computer Programming Language Pascal.* International Organization for Standardization (ISO), May 1980.

[65] K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, New York, second edition, 1978.

[66] S.C. Johnson. Yacc – yet another compiler-compiler. Technical Report No. 23, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[67] J.B. Johnston. The Contour Model of block structured processes. In J.T. Tou and P. Wegner (editors), *Proceedings of the Symposium on Data Structures in Programming Languages*, pages 55–82, 1971. *ACM SIGPLAN Notices*, Volume 6, Number 2, February 1971.

[68] U. Kastens, B. Hutte and E. Zimmermann. *GAG: A Practical Compiler Generator*, Volume 141 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1982.

[69] E. Keiser. Ack-c reference manual. Technical report, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 12^{th} September 1983.

[70] E. Keiser. Ack description file reference manual. Technical report, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 1983.

[71] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[72] R.B. Kieburtz. Steps toward verifiable programs. Technical report, Department of Computer Science, State University of New York, Stony Brook, New York, November 1972.

[73] V. Kini, D.F. Martin and A. Stoughton. Tools for testing denotational semantic definitions of programming languages. Technical Report ISI/RR-83-112, Information Sciences Institute, Marina del Ray, California, U.S.A., May 1983.

[74] C. Kirchner, H. Kirchner and J. Meseguer. Operational semantics of OBJ-3. In T. Lepistö and A. Salomaa (editors), *Automata, Languages and Programming*, pages 287–301. Springer-Verlag, Berlin, 1988. Volume 317 of *Lecture Notes in Computer Science*.

[75] K. Koskimies, K.-J. Räihä and M. Sarjakoski. Compiler construction using attribute grammars. *ACM SIGPLAN Notices*, Volume 17, Number 6, pages 153–159, June 1982. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Boston, Massachusetts, 23-25 June, 1982.

[76] P. Lee. *Realistic Compiler Generation*. Foundations of Computing Series. The MIT Press, Cambridge, Massachusetts, 1989.

[77] T. Lehmann and J. Loeckx. The specification language OBSCURE. In D. Sannella and A. Tarlecki (editors), *Recent Trends in Data Type Specifications: 5th Workshop on Specification of Abstract Data Types*, pages 131–153. Springer-Verlag, Berlin, 1988. Volume 332 of *Lecture Notes in Computer Science*.

[78] C.-W. Lerman and J. Loeckx. OBSCURE a new specification language. In H.-J. Kreowski (editor), *Recent Trends in Data Type Specifications: 3rd Workshop on Theory and Applications of Abstract Data Types*, pages 28–30. Springer-Verlag, Berlin, 1985. Volume 116 of *Informatik-Fachberichte*.

[79] M.E. Lesk. Lex – a lexical analyser generator. Technical Report Computer Science No. 59, Bell Laboratories, Murray Hill, New Jersey, October 1975.

[80] W.R. Mallgren. *Formal Specification of Interactive Graphics Programming Languages*. MIT Press, Cambridge, Massachusetts, 1983.

[81] J. Malpas. *Prolog: A Relational Language and its Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[82] C.D. Marlin. A model for data control in the programming language Pascal. In *Proceedings of the Australian Colleges of Advanced Education Computing Conference*, pages 293–306, Adelaide, August 1977.

[83] C.D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, Volume 95 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[84] C.D. Marlin and D.H. Freidel. A model for communication in programming languages with buffered message passing. Technical Report 83–09, Department of Computer Science, The University of Iowa, Iowa City, Iowa, November 1983.

[85] C.D. Marlin and M.J. Oudshoorn. Using abstract data types in a model of the data control aspect of programming languages. *Australian Computer Science Communications*, Volume 7, Number 1, pages 19–1 – 19–10, February 1985.

[86] C.D. Marlin, M.J. Oudshoorn and D.H. Freidel. A model of communication in Ada using shared data abstractions. In S.G. Akl, F. Fiala and W.W. Koczkodaj (editors), *Advances in Computing and Information – ICCI'90*, pages 443–452. Springer-Verlag, Berlin, 1990. Volume 468 of *Lecture Notes in Computer Science.*

[87] C.D. Marlin, M.J. Oudshoorn and D.H. Freidel. A model of intertask communication in Ada. In *Proceedings of the 1990 International Conference on Computing and Information*, pages 434–440, May 1990.

[88] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart and M.I. Levin. *LISP 1.5 Programmers Manual.* MIT Press, Cambridge, Massachusetts, second edition, 1965.

[89] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce and R. Thomas. SISAL: Streams and iteration in a single assignment language; Language Reference Manual Version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, California, March 1985.

[90] I. Mearns. A denotational semantics for concurrent Ada programs. Technical Report UMCS–83-10-1, Department of Computer Science, University of Manchester, Manchester, England, October 1983.

[91] E. Meiling. A comparative study of CHILL and Ada on the basis of denotational descriptions. *ACM Ada Letters*, Volume III, Number 4, pages 78–90, January – February 1984.

[92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[93] K.W. Miller. *Programming in Vision Research Using Pixelspaces, A Data Abstraction.* Ph.D. thesis, Department of Computer Science, The University of Iowa, Iowa City, Iowa, July 1983. Available as Technical Report 83-04.

[94] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[95] J.C. Mitchell, W. Maybury, R. Sweet and J.R. Hertz Jr. Mesa language manual. Technical report, Office Systems Division, Xerox Corporation, Palo Alto, California, June 1984. Version 11.0.

[96] B.P. Molinari and C.W. Johnson. Generation of symbol processing modules. Technical Report TR-CS-87-02, Department of Computer Science, The Australian National University, Canberra, Australian Capital Territory, June 1987.

[97] R. Morrison. PS-algol reference manual. Technical Report 12, Department of Computational Science, University of St. Andrews, St. Andrews, Scotland, February 1988. Fourth Edition.

[98] P.D. Mosses. *Mathematical semantics and compiler generation.* Ph.D. thesis, Oxford University, London, England, 1975.

[99] P.D. Mosses. SIS – semantics implementation system. Technical Report DAIMI PB-217, Computer Science Department, Aarhus University, Aarhus, Denmark, 1979.

[100] P. Naur. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, Volume 3, pages 299–314, 1960.

[101] T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, Volume 11, Number 4, pages 650–665, October 1989.

[102] R.D. Nicola, A. Martelli and U. Montanari. Communication through message passing or shared memory: A formal comparison. In *Proceedings of the Second*

International Conference on Distributed Computing Systems, pages 513–522, Washington D.C., 1981. IEEE Computer Society Press.

[103] M.J. Oudshoorn and C.D. Marlin. Describing data control in programming languages. In *IEEE International Conference on Computer Languages'88*, pages 100–109, Miami Beach, Florida, October 9–13 1988.

[104] M.J. Oudshoorn and C.D. Marlin. Language definition and implementation. *Australian Computer Science Communications*, Volume 11, Number 1, pages 26–36, February 1989.

[105] M.J. Oudshoorn and C.D. Marlin. Describing the semantics of parallel programming languages using shared data abstractions. Technical Report 91-03, Department of Computer Science, The University of Adelaide, Adelaide, South Australia, May 1991.

[106] M.J. Oudshoorn and C.D. Marlin. A layered, operational model of data control in programming languages. *Computer Languages*, Volume 16, Number 2, pages 147–165, 1991.

[107] M.J. Oudshoorn, K.J. Ransom and C.D. Marlin. Abstract data types: Converting from sequential to parallel. In P.A. Bailes (editor), *Engineering Safe Software*, pages 285–298. Australian Computer Society, Sydney, New South Wales, July 1991. Proceedings of the *1991 Australian Software Engineering Conference.*

[108] S. Owicki. Verifying concurrent programs with shared data classes. In *Formal Descriptions of Programming Concepts*, pages 279–299. North-Holland Publishing Company, Amsterdam, 1978.

[109] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, Volume 6, pages 319–340, 1976.

[110] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, Volume 19, Number 5, pages 279–285, May 1976.

[111] P. Padawitz. Parameter preserving data type specifications. In H. Ehrig, C. Floyd, M. Nivat and J. Thatcher (editors), *Mathematical Foundations of Software Development*, pages 323–341. Springer-Verlag, Berlin, 1985. Volume 185 of *Lecture Notes in Computer Science.*

[112] F.G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[113] L. Paulson. *A Compiler Generator for Semantic Grammars.* Ph.D. thesis, Stanford University, Stanford, California, 1981.

[114] L. Paulson. A semantics-directed compiler generator. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 224–232, Albuquerque, New Mexico, January 1982.

[115] A. Poigné. Error handling for parameterized data types. In H.-J. Kreowski (editor), *Recent Trends in Data Type Specification. Third Workshop on Theory and Applications of Abstract Data Types – Selected Papers*, pages 224–239. Springer-Verlag, Berlin, 1985. Volume 116 of *Informatik-Fachberichte.*

[116] T.W. Pratt. *Programming Languages: Design and Implementation.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, second edition, 1984.

[117] K. Ramamritham and R.M. Keller. Specifying and proving properties of sequential processes. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 374–382, 1981.

[118] S.P. Reiss. Generation of compiler symbol processing mechanisms from specifications. *ACM Transactions of Programming Languages and Systems*, Volume 5, Number 2, pages 127–163, April 1983.

[119] J. Röhrich. Private communication, August 1985.

[120] A.W. Roscoe. Denotational semantics for Occam. In S.D. Brookes, A.W. Roscoe and G. Wiskel (editors), *Seminar on Concurrency*, pages 306–329, Berlin, 1984. Springer-Verlag. Volume 197 of *Lecture Notes in Computer Science.*

[121] G.A. Rose and J. Welsh. Formatted programming languages. *Software – Practice and Experience*, Volume 11, Number 7, pages 651–669, July 1981.

[122] B.K. Rosen. Tree–manipulating systems and Church–Rosser theorems. *Journal of the ACM*, Volume 20, Number 1, pages 160–187, January 1973.

[123] D.J. Salomon and C.V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 170–178, Portland, Oregon, June 21–23 1989. *ACM SIGPLAN Notices*, Volume 24, Number 7, July 1989.

[124] J.G. Sanderson. *A Relational Theory of Computing*, Volume 82 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[125] J.G. Sanderson. Relator calculus. *Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, Volume 10, Number 1, pages 63–100, 1990.

[126] R.D. Schlichting and F.B. Schneider. Using message passing for distributed programming: Proof rules and discipline. Technical Report 82-5, Department of Computer Science, University of Arizona, Tucson, Arizona, June 1982.

[127] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon Inc., Newton, Massachusetts, 1986.

[128] S. Sidhar. An implementation of OBJ2: An object-oriented language for abstract program specification. In K.V. Nori (editor), *Foundations of Software Technology and Theoretical Computer Science*, pages 81–95. Springer-Verlag, Berlin, 1986. Volume 241 of *Lecture Notes in Computer Science.*

[129] S.K. Skedzielewski and J. Glauert. IF1 – an intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, California, July 1985.

[130] S.K. Skedzielewski and M.L. Welcome. Data flow graph optimization in IF1. In J.P. Jouannaud (editor), *Functional Programming Languages and Computer Architecture*, pages 17–34. Springer-Verlag, September 1985. Volume 201 of *Lecture Notes in Computer Science.*

[131] C.L. Smith. *A Formal Analysis of Name Accessing in Programming Languages.* Ph.D. thesis, Iowa State University, Ames, Iowa, 1975.

[132] G.L. Steele Jr. *Common LISP: The Language.* Digital Press, Burlingham, Massachusetts, 1984.

[133] J.W. Stevenson. Amsterdam compiler kit – Pascal reference manual. Technical report, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 4^{th} January 1983.

[134] J.E. Stoy. *Denotational Semantics.* MIT Press, Cambridge, Massachusetts, 1977.

[135] B. Stroustrup. *The C++ Programming Language.* Addison–Wesley, Reading, Massachusetts, 1987.

[136] A.S. Tanenbaum, J.W. Stevenson, E.G. Keizer and H. van Staveren. A practical tool kit for making portable compilers. Technical Report 74, Vrije Universiteit, Amsterdam, 1983.

[137] A.S. Tanenbaum, H. van Staveren, E.G. Keizer and J.W. Stevenson. Description of a machine architecture for the use with block structured languages. Technical Report IR-81, Vrije Universiteit, Amsterdam, August 1983.

[138] R.D. Tennent. A denotational definition of the programming language Pascal. Technical report, Programming Research Group, Oxford University, Oxford, England, April 1978.

[139] J.W. Thatcher, E.G. Wagner and J.B. Wright. Data type specification: Parameterization and the power of specification techniques. *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 4, pages 711–732, October 1982.

[140] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein and W. Kirchgässner. *An Attribute Grammar for the Semantic Analysis of Ada*, Volume 139 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1982.

[141] U.S. Department of Defense. *The Programming Language Ada Reference Manual, ANSI/MIL-STD-1815A-1983.* United States Department of Defense, Washington, D.C., 1983.

[142] H. van Staveren. The table driven code generator from the Amsterdam compiler kit. Technical report, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 1983.

[143] A. van Wijngaarden. Recursive definition of syntax and semantics. In T.B. Steel (editor), *Formal Language Description Languages for Computer Programming*, pages 13–24. North-Holland Publishing Company, Amsterdam, 1966.

[144] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck and C.H.A. Koster. Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, Volume 14, Number 2, pages 79–218, February 1969.

[145] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meerens and R.G. Fisher (editors). *Revised Report on the Algorithmic Language Algol 68.* Springer-Verlag, Berlin, 1976.

[146] T.N. Vickers. Quokka: A translator generator using denotational semantics. *The Australian Computer Journal*, Volume 18, Number 1, pages 9–17, February 1986.

[147] M. Wand. A semantic prototyping system. *ACM SIGPLAN Notices*, Volume 19, Number 6, pages 213–221, June 1984. Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, Montreal, Canada, 17–22 June, 1984.

[148] D.A. Watt. An extended attribute grammar for Pascal. *ACM SIGPLAN Notices*, Volume 14, Number 2, pages 60–74, February 1979.

[149] D.A. Watt, B.A. Wichmann and W. Findlay. *Ada. Language and Methodology.* Prentice-Hall, Engelwood Cliffs, New Jersey, 1987.

[150] P. Wegner. Data structure models for programming languages. In J.T. Tou and P. Wegner (editors), *Proceedings of the Symposium on Data Structures in Programming Languages*, pages 1–54, 1971. *ACM SIGPLAN Notices*, Volume 6, Number 2, February 1971.

[151] J. Welsh and P. Bailes. Modula-2 standardisation: The go-betweens' tale. *The MODUS Quarterly*, Volume 7, pages 3–6, February 1987.

[152] J. Welsh, W.J. Sneeringer and C.A.R. Hoare. Ambiguities and insecurities in Pascal. *Software – Practice and Experience*, Volume 7, pages 685–696, 1977.

[153] A.L. Wendelborn. *Data Flow Implementations of a Lucid-like Programming Language.* Ph.D. thesis, Department of Computer Science, University of Adelaide, Adelaide, South Australia, 1985. Available as Technical Report 85-02.

[154] J.M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, Volume 9, Number 1, pages 1–24, January 1987.

[155] N. Wirth. The programming language Pascal. *Acta Informatica*, Volume 1, Number 1, pages 35–63, January 1971.

[156] N. Wirth. Design and implementation of Modula. *Software – Practice and Experience*, Volume 7, Number 1, pages 67–84, January–February 1977.

[157] N. Wirth. Modula: A language for modular multiprogramming. *Software – Practice and Experience*, Volume 7, Number 1, pages 3–35, January–February 1977.

[158] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, Volume 20, Number 11, pages 822–823, November 1977.

[159] N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, third, corrected edition, 1985.

[160] N. Wirth. From Modula to Oberon *and* The programming language Oberon. Technical Report 82, Institut für Informatik, ETH-Zentrum, Zurich, Switzerland, September 1987.

[161] X3 Secretariat. Draft programming language C. Technical report, ISO, Washington D.C., February 14 1986. Document Number X3J11/86-017.

[162] X3 Secretariat. Working draft extended Pascal standard. Technical report, ISO, Washington D.C., 1986. Document Number X3J9/86-004.

[163] S.N. Zilles. An introduction to data algebras. In D. Bjørner (editor), *Abstract Software Specifications. 1979 Copenhagen Winter School Proceedings*, pages 248–292, Berlin, 22nd January – 2nd February 1979. Springer-Verlag. Volume 86 of *Lecture Notes in Computer Science.*

[164] S.N. Zilles. Types, algebras and modeling. *ACM SIGPLAN Notices*, Volume 16, Number 1, pages 207–209, January 1981. Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, Pingree Park, Colorado, June 23–26, 1980.