Chapter 12
ALGEBRAIC SEMANTICS

The formal semantic techniques we have studied so far include denotational semantics, whose mathematical foundations lie in recursive function theory, and axiomatic semantics, whose foundations depend on predicate logic. In this chapter we study algebraic semantics, another formalism for semantic specification whose foundations are based on abstract algebras. Algebraic semantics involves the algebraic specification of data and language constructs. The basic idea of the algebraic approach to semantics is to name the different sorts of objects and the operations on the objects and to use algebraic axioms to describe their characteristic properties.

The methodology of algebraic semantics is customarily used to specify abstract data types (ADTs). The basic principle in specifying an ADT involves describing the logical properties of data objects in terms of properties of operations (some of which may be constants) that manipulate the data. The actual representation of the data objects and the implementations of the operations on the data are not part of the specification. For instance, we specify the abstract type whose values are stacks by defining the properties of the operations that push or pop items from the stacks, avoiding a description of a physical representation of the objects that serve as stacks.

In this chapter we introduce the basic ideas of algebraic specifications (syntax) and the corresponding algebras (semantics) that serve as models of specifications. As we will see, algebraic specifications extend from low-level objects, such as truth values with Boolean operations, through high-level objects, such as programs with operations to perform type checking and to interpret source code. Algebraic semantics is a broad field of study, and in this brief overview we can only suggest the underlying mathematical foundations. Some of the fundamental notions developed here will be used in the next chapter when we investigate our final approach to semantics, action semantics.

12.1 CONCEPTS AND EXAMPLES

Before exploring examples, we introduce some of the vocabulary of algebraic specification. The types in a programming language serve to classify the data processed by programs. Here we refer to types as **sorts**. An algebraic specification defining one or more sorts contains two parts: the **signature** and the **equations** (or axioms).

Definition: A **signature** Σ of an algebraic specification is a pair <Sorts, Operations> where Sorts is a set containing names of sorts and Operations is a family of function symbols indexed by the functionalities of the operations represented by the function symbols. ▮

We use the terms "functions" and "operations" interchangeably, but when considering specifications, these terms refer to formal function symbols. The set of operations in a specification provides the syntax of the functions that are defined on the sorts of data. Suppose we want to specify an abstract type whose values are lists of integers. We provide three sorts in the specification:

Sorts = { Integer, Boolean, List }.

The elements of Sorts are only names; we can assume nothing about the properties of these sorts. The set of operations may include the function symbols given below with their signatures:

zero	: Integer
one	: Integer
plus (_ , _)	: Integer, Integer → Integer
minus (_ , _)	: Integer, Integer → Integer
true	: Boolean
false	: Boolean
emptyList	: List
cons (_ , _)	: Integer, List → List
head (_)	: List → Integer
tail (_)	: List → List
empty? (_)	: List → Boolean
length (_)	: List → Integer

The family of operations can be decomposed into eight sets of function symbols indexed by the domain-codomain constraints on the functions. We list several of the sets of operations in the family:

$\mathrm{Opr}_{\mathrm{Boolean}}$ = { true, false }

$\mathrm{Opr}_{\mathrm{Integer,Integer}\rightarrow\mathrm{Integer}}$ = { plus, minus }

$\mathrm{Opr}_{\mathrm{List}\rightarrow\mathrm{Integer}}$ = { head, length }

Other sets of operations are indexed by Integer, List, (Integer,List→List), (List→List), and (List→Boolean). Observe that operations with no domain represent constants of a particular sort—for example, zero, one, true, false, and emptyList. The signature of a specification can be compared with the declarations in a program—a specification defines the kinds of objects to which names will refer. The signature shown above tells us how we may use identifiers such as List, cons, and length but does not describe the behavior of the corresponding functions.

The equations in a specification act to constrain the operations in such a way as to indicate the appropriate behavior for the operations. They serve as axioms specifying an algebra, similar to the properties of associativity and commutativity of operations that we associate with abstract algebras in mathematics. Equations may involve variables representing arbitrary values from the various sorts in the specification. The variables in an equation are universally quantified implicitly. Listed below are several equations (axioms) that may appear in a specification of lists.

```
head (cons (m, s))      = m
empty? (emptyList)      = true
empty? (cons (m, s))    = false
```

The first equation stands for the closed assertion:

$\forall$m:Integer, $\forall$s:List [head (cons (m, s)) = m].

Since indexed sets can challenge our tolerance of notation, algebraic specifications are commonly represented using a module-like syntactic structure that encapsulates the pertinent information concerning the signature and the equations of the specification. We have already seen how the family of operations will be specified when we used the “function header” notation above:

cons (_ , _) : Integer,List → List.

The syntax of the function symbol is spelled out as a pattern, with underscores representing the parameter positions and the Cartesian product indicated by a comma forming the domain. These notational conventions have become common practice in algebraic specifications.

Another advantage of the module representation of algebraic specifications is that it lends itself to decomposing definitions into relatively small components. We break the specification of lists of integers into three smaller modules for integers, Boolean values, and then lists, using a mechanism to import the signature and equations of one module into another. We view an algebraic specification as a sequence of modules, so that when one module imports another module, the sorts and functions in the signature can be used in the importing module. Relative to this importing mechanism, we define sorts and functions to be either exported or hidden. Hidden symbols are visible only in the module where they are first defined. Later we see that modules can be parameterized to define generic abstract data types. With parameterized specifications, certain portions of the module are left unspecified until the module is instantiated by specifying values for the formal parameters.

A Module for Truth Values

We now turn our attention to a module that gives an algebraic specification of truth values.

```
module Booleans
    exports
        sorts Boolean
        operations
            true               :  Boolean
            false              :  Boolean
            errorBoolean       :  Boolean
            not ( _ )          :  Boolean → Boolean
            and ( _ , _ )      :  Boolean, Boolean → Boolean
            or ( _ , _ )       :  Boolean, Boolean → Boolean
            implies ( _ , _ )  :  Boolean, Boolean → Boolean
            eq? ( _ , _ )      :  Boolean, Boolean → Boolean
    end exports

    operations
        xor ( _ , _ ) : Boolean, Boolean → Boolean

    variables
        b, b1, b2 : Boolean

    equations
    [B1]    and (true, b)         =  b
    [B2]    and (false, true)     =  false
    [B3]    and (false, false)    =  false
    [B4]    not (true)            =  false
    [B5]    not (false)           =  true
    [B6]    or (b1, b2)           =  not (and (not (b1), not (b2)))
    [B7]    implies (b1, b2)      =  or (not (b1), b2)
    [B8]    xor (b1, b2)          =  and (or (b1, b2), not (and (b1, b2)))
    [B9]    eq? (b1, b2)          =  not (xor (b1, b2))
end Booleans
```

The sort Boolean has two “normal” constant values, true and false, and an error value errorBoolean. We discuss the handling of errors in conjunction with the next module that specifies natural numbers. The functions not, and, or, and eq? are exported, whereas the function xor (exclusive or) is hidden. The module has no hidden sorts. Remember that the variables in equations are universally quantified implicitly, so that equation [B1] represents the axiom:

∀b:Boolean [and (true, b) = b].

The equations in a specification may allow several variations that result ultimately in the same definition. For example, the semantics of or can be specified directly by

```
or (true, true)  = true
or (true, false) = true
or (false, b)    = b.
```

Although these different definitions often suggest different evaluation strategies, the equations in a specification are purely declarative and do not imply any particular evaluation order. The xor is defined in terms of and, or, and not. In order to illustrate hidden functions, we have elected not to make xor public. The eq? function is defined as the logical negation of xor. A direct definition of eq? is also possible.

Module Syntax

Before turning our attention to more sample modules, we examine the structure of a typical module. Each of the components specified below may be omitted in defining a particular module.

```
module <module-name>
    imports
        <list of modules>
    parameters <parameter name>
        sorts <sort names>
        operations <function symbols with their signatures>
        variables <list of variables and their sorts>
        equations <unconditional and conditional equations>
    end <parameter name>
    exports
        sorts <list of public sorts>
        operations <list containing signatures of public function symbols>
    end exports
    sorts <hidden sort names>
    operations
        <hidden function symbols with their signatures>
    variables
        <list of variables and their sorts>
    equations
        <unconditional and conditional equations>
end <module-name>
```

The second section contains parameters that are defined in terms of formal parameter symbols. The actual values (arguments) are supplied when the module is instantiated by naming the parameterized module and supplying already defined sorts and operations for each of the formal sort names and function symbols in the parameter. The functionality (syntax) of the actual operations must agree with the formal parameters, and the argument functions must satisfy the equations in the parameters section that specify properties of the formal parameters. We have shown the format for a single parameter, but multiple parameters are also possible. When modules are imported, items in them may be renamed. We show the syntax for renaming later in the section.

The syntax of function application may be represented using several forms, but in this chapter we rely on ordinary prefix notation with parentheses delimiting the arguments. This notation eliminates the need for precedence rules to disambiguate the order of execution of the corresponding operations. In the next chapter we consider some variations on this notation for function application. Functions can also return multiple values as tuples—for example:

$$h : S_1 \rightarrow S_2, S_3.$$

Tupled outputs are a notational convenience and can be replaced by a single sort. These details are left as an exercise.

Equations specifying the properties of operations can be unconditional, as in the Booleans module, or conditional. A **conditional equation** has the form

$$\text{lhs} = \text{rhs}\ \textit{when}\ \text{lhs}_1 = \text{rhs}_1, \text{lhs}_2 = \text{rhs}_2, \ldots, \text{lhs}_n = \text{rhs}_n.$$

Finally, we mention that modules cannot be nested.

A Module for Natural Numbers

In the rest of this section, we give more sample modules to illustrate the ideas introduced above, concentrating on specifications that will be needed to define the semantics of Wren. The next module specifies the natural numbers, containing constant function symbols 0, 1, 10 and errorNatural, a function succ used to construct terms for the specification, the numeric operations add, sub, mul, div, and the predicate operations eq?, less?, and greater?. An exercise asks the reader to add an exponentiation operation exp to Naturals. This function is used when Naturals is imported by a module called Strings.

```
module Naturals
    imports Booleans
    exports
        sorts Natural
        operations
            0                     :  Natural
            1                     :  Natural
            10                    :  Natural
            errorNatural          :  Natural
            succ ( _ )            :  Natural  → Natural
            add ( _ , _ )         :  Natural, Natural → Natural
            sub ( _ , _ )         :  Natural, Natural → Natural
            mul ( _ , _ )         :  Natural, Natural → Natural
            div ( _ , _ )         :  Natural, Natural → Natural
            eq? ( _ , _ )         :  Natural, Natural → Boolean
            less? ( _ , _ )       :  Natural, Natural → Boolean
            greater? ( _ , _ )    :  Natural, Natural → Boolean
    end exports

    variables
        m, n : Natural

    equations
    [N1]   1                        = succ (0)
    [N2]   10                       = succ (succ (succ (succ (succ (succ (
                                          succ (succ (succ (succ (0))))))))))
    [N3]   add (m, 0)               = m
    [N4]   add (m, succ (n))        = succ (add (m, n))
    [N5]   sub (0, succ(n))         = errorNatural
    [N6]   sub (m, 0)               = m
    [N7]   sub (succ (m), succ (n)) = sub (m, n)
    [N8]   mul (m, 0)               = 0                  when m≠errorNatural
    [N9]   mul (m, 1)               = m
    [N10]  mul (m, succ(n))         = add (m, mul (m, n))
    [N11]  div (m, 0)               = errorNatural
    [N12]  div (0, succ (n))        = 0                  when n≠errorNatural
    [N13]  div (m, succ (n))            = if (less? (m, succ (n)),
                                                0,
                                                succ(div(sub(m,succ(n)),succ(n))))
    [N14]  eq? (0, 0)                   = true
    [N15]  eq? (0, succ (n))            = false          when n≠errorNatural
    [N16]  eq? (succ (m), 0)            = false          when m≠errorNatural
    [N17]  eq? (succ (m), succ (n))     = eq? (m, n)
    [N18]  less? (0, succ (m))          = true           when m≠errorNatural
    [N19]  less? (m, 0)                 = false          when m≠errorNatural
    [N20]  less? (succ (m), succ (n))   = less? (m, n)
    [N21]  greater? (m, n)              = less? (n, m)
end Naturals
```

Each sort will contain an error value to represent the result of operations on values outside of their normal domains. We assume that all operations propagate errors, so that, for example, the following properties hold:

```
    succ (errorNatural)            = errorNatural
    mul (succ (errorNatural), 0)   = errorNatural
    or (true, errorBoolean)        = errorBoolean
    eq? (succ(0), errorNatural)    = errorBoolean.
```

Propagating errors in this manner requires that some equations have conditions that restrict the operations to nonerror values. Without these conditions, the entire sort reduces to the error value, as witnessed by a deduction using [N8] and ignoring the condition:

0 = mul(succ(errorNatural),0) = mul(errorNatural,0) = errorNatural.

Without the condition on [N8], all the objects in Natural can be shown to equal errorNatural:

succ(0) = succ(errorNatural) = errorNatural,

succ(succ(0)) = succ(errorNatural) = errorNatural,

and so on.

The equations that require conditions to avoid this sort of inconsistency—namely, [N8], [N12], [N15], [N16], [N18], and [N19] in the Naturals module—are those in which the variable(s) on the left disappear on the right. As a notational convention to enhance readability, we use n≠errorNatural for eq? (n,errorNatural) = false and similar abbreviations for the other sorts.

The module Naturals has no equation defining properties for the succ function. This operation is called a **constructor**, since together with the constant 0, succ can be used to construct representations of the values that form the natural numbers. In a model of this specification, we assume that values are equal only when their identity can be derived from the equations. Since there are no equations for succ, 0 does not equal succ(0), which does not equal succ(succ(0)), and so forth. So the terms 0, succ(0), succ(succ(0)), … can be viewed as characterizing the natural numbers, the objects defined by the module.

The element errorNatural can be derived from the equations in two cases: subtracting a number from another number smaller than itself and dividing any number by zero. Thus we say that the terms that can be generated in the natural numbers specification consist of the values 0, succ(0), succ(succ(0), succ(succ(succ(0))), … and errorNatural. This set serves as the universe of values or the **carrier set** for one of the algebras that acts as a model of the specification. We will define this so-called term algebra model carefully when we discuss the mathematical foundations of algebraic specifications in the next section.

This method of interpreting the meaning of the equations is known as **initial algebraic semantics** . In an initial model, the equality of items can be proved only as a direct result of equations in the module; otherwise they are never considered to be the same. This characteristic is called the **no confusion** property. The **no junk** property of the initial model says that all terms in the carrier set of a model of the specification correspond to terms generated from the signature of the module. We will examine the no confusion and no junk properties again in the next section when we discuss the construction of an initial algebra and describe its properties.

The constant functions 1 and 10 are convenient renamings of particular natural numbers for easy reference outside of the module. Addition is defined recursively with the second operand being decremented until it eventually reaches 0, the base case. The other operations, sub, mul, and div are defined recursively in similar ways. The rule [N9] is redundant since its values are included in [N10]. The div equations introduce the built-in polymorphic function *if*, which is used to determine when div has reached the base case (the dividend is less than the divisor).

```
div (m, succ (n)) =  if ( less? (m, succ (n)),
                          0,
                          succ (div (sub (m, succ (n)), succ (n))))
```

A generic *if* operation cannot be specified by an algebraic specification itself since its arguments range over all possible sorts. However, it is always possible to eliminate the *if* by writing multiple conditional equations. For example,

```
div (m, succ (n))  = 0                  when less? (m, succ (n)) = true
div (m, succ (n))  = succ ( div (sub (m, succ (n)), succ (n))))
                                        when less? (m, succ (n)) = false
div (m, succ (n))  = errorNatural       when less? (m, succ (n)) = errorBoolean.
```

Given this equivalence, we will continue to use *if* as a notational convenience without sacrificing the underlying foundations of algebraic semantics. Observe that using *if* requires that the module Booleans be imported to provide truth values.

A Module for Characters

The Characters module presented below defines an underlying character set adequate for Wren identifiers. Although this character set is limited to digits and lowercase characters, the module can be easily extended to a larger character set.

```
module Characters
    imports  Booleans, Naturals
    exports
        sorts Char
        operations
            eq? ( _ , _ )  :  Char, Char → Boolean
            letter? ( _ )  :  Char → Boolean
            digit? ( _ )   :  Char → Boolean
            ord ( _ )      :  Char → Natural
            char-0         :  Char
            char-1         :  Char
              :                 :
            char-9         :  Char
            char-a         :  Char
            char-b         :  Char
            char-c         :  Char
              :                 :
            char-y         :  Char
            char-z         :  Char
            errorChar      :  Char
    end exports

    variables
        c, c1, c2  : Char

    equations
    [C1]    ord (char-0)  =  0
    [C2]    ord (char-1)  =  succ (ord (char-0))
    [C3]    ord (char-2)  =  succ (ord (char-1))
      :           :                 :
    [C10]   ord (char-9)  =  succ (ord (char-8))
    [C11]   ord (char-a)  =  succ (ord (char-9))
    [C12]   ord (char-b)  =  succ (ord (char-a))
    [C13]   ord (char-c)  =  succ (ord (char-b))
      :           :                 :
    [C35]   ord (char-y)  =  succ (ord (char-x))
    [C36]   ord (char-z)  =  succ (ord (char-y))
    [C37]   eq? (c1, c2)  =  eq? (ord (c1), ord (c2))
    [C38]   letter? (c)   =  and (not (greater? (ord (char-a), ord (c))),
                                        not (greater? (ord (c), ord (char-z))))
    [C39]   digit? (c)    =  and (not (greater? (ord (char-0), ord (c))),
                                        not (greater? (ord (c) ord (char-9))))
end Characters
```

Observe that the equation "ord (char-9) = 9" cannot be derived from the equations in the module, since 9 is not a constant defined in Naturals. Rather than add the constants 2 through 9 and 11 through 35 to Naturals (for this character set), we rely on the repeated application of the successor function to define the ordinal values. The ord function here does not produce ascii codes as in Pascal but simply gives integer values starting at 0. Note that eq? refers to two different operations in [C37]. The first represents the equality operation on the characters currently being defined, and the second symbolizes equality on the natural numbers imported from Naturals. The sorts of the arguments determine which of the overloaded eq? operations the function symbol represents.

A Parameterized Module and Some Instantiations

The next example shows a parameterized module for homogeneous lists where the type of the items in the lists is specified when the module is instantiated.

```
module Lists
    imports  Booleans, Naturals

    parameters Items
        sorts Item
        operations
            errorItem  :  Item
            eq?        :  Item, Item → Boolean
        variables
            a, b, c : Item
        equations
            eq? (a,a)  = true              when a≠errorItem
            eq? (a,b)  = eq? (b,a)
            implies (and (eq? (a,b), eq? (b,c)), eq? (a,c)) = true
                                           when a≠errorItem, b≠errorItem, c≠errorItem
    end Items

    exports
        sorts List
        operations
            null               :  List
            errorList          :  List
            cons ( _ , _ )     :  Item, List  → List
            concat ( _ , _ )   :  List, List  → List
            equal? ( _ , _ )   :  List, List  → Boolean
            length ( _ )       :  List → Natural
            mkList ( _ )       :  Item → List
        end exports

    variables
        i, i1, i2 : Item
        s, s1, s2  : List

    equations
    [S1]    concat (null, s)            =  s
    [S2]    concat (cons (i, s1), s2)   =  cons (i, concat (s1, s2))
    [S3]    equal? (null, null)         =  true
    [S4]    equal? (null, cons (i, s))  =  false          when s≠errorList, i≠errorItem
    [S5]    equal? (cons (i, s), null)  =  false          when s≠errorList, i≠errorItem
    [S6]    equal? (cons (i1, s1), cons (i2, s2)) = and (eq? (i1, i2), equal? (s1, s2))
    [S7]    length (null)               =  0
    [S8]    length (cons (i, s))        =  succ (length (s))     when  i≠errorItem
    [S9]    mkList (i)                  =  cons (i, null)
end Lists
```

The three parameters for Lists define the type of items that are being joined, an error value, and an equality test required for comparing the items. The equations in the parameter section ensure that the operation associated with the formal symbol eq? is an equivalence relation. The symbols that act as parameters are unspecified until the module is instantiated when imported into another module. The operation null is a constant that acts as a constructor representing the empty list, and cons is a constructor function, having no defining equations, that builds nonempty lists. The structures formed by applications of the constructor functions represent the values of the sort List.

The length function is defined recursively with the base case treating the null list and the other case handling all other lists. The operation mkList is a convenience function to convert a single item into a list containing only that item. Since cons and null are exported, the user can do this task directly, but having a named function improves readability.

The process of renaming is illustrated in the following example of a list of integers, called Files, that will be used for input and output in the specification of Wren. Sometimes sorts and operations are renamed purely for purposes of documentation. Items that are not renamed on import retain their original names. For example, the constructor for Files will still be cons. See the further readings at the end of this chapter for more on these issues.

```
module Files
    imports Booleans, Naturals,
        instantiation of Lists
            bind Items using Natural for Item
                       using errorNatural for errorItem
                       using eq? for eq?
            rename   using File for List
                     using emptyFile for null
                     using mkFile for mkList
                     using errorFile for errorList
    exports
        sorts File
        operations
            empty? ( _ ) : File → Boolean
    end exports

    variables
        f : File

    equations
    [F1]    empty? (f) = equal? (f, emptyFile)
end Files
```

The identifiers created by renaming are exported by the module as well as the identifiers in the exports section. Note that we extend the instantiated imported module Lists with a new operation empty?. The Strings module, which is used to specify identifiers in Wren, also contains an instantiation of Lists.

```
module Strings
    imports Booleans,Naturals, Characters,
            instantiation of Lists
                    bind Items using Char for Item
                               using errorChar for errorItem
                               using eq? for eq?
                    rename using String for List
                           using nullString for null
                           using mkString for mkList
                           using strEqual for equal?
                           using errorString for errorList
    exports
        sorts String
        operations
            string-to-natural ( _ ) : String → Boolean, Natural
    end exports
    variables
        c : Char
        b : Boolean
        n : Natural
        s : String
    equations
    [Str1]  string-to-natural (nullString) = <true,0>
    [Str2]  string-to-natural (cons (c, s))=
                if ( and (digit? (c), b),
                    <true, add (mul (sub (ord (c), ord (char-0)),
                                            exp (10, length (s))), n)>,
                    <false, 0>)
                            when <b,n> = string-to-natural (s)
end Strings
```

The string-to-natural function returns a pair: The first value is a truth value that indicates whether the conversion was successful, and the second value is the numeric result. We introduced the constant 10 in Naturals to make this specification more readable. The operation exp is added to Naturals in an exercise.

A Module for Finite Mappings

The final modules of this section are Mappings and an instantiation of Mappings. A mapping associates a domain value of some sort with an item taken from some range (codomain) sort. Both the domain and range sorts are specified by parameters and determined at the time of instantiation. We use two mappings later in an algebraic specification of Wren. The type checking module associates Wren types with variable names modeling a symbol table, and the evaluation (or execution) module associates numeric values with variable names, modeling a store. Both of these sorts result from instantiations of Mappings.

```
module Mappings
    imports Booleans
    parameters Entries
            sorts Domain, Range
            operations
                equals ( _ , _ )  :  Domain, Domain → Boolean
                errorDomain       :  Domain
                errorRange        :  Range
            variables
                a, b, c : Domain
            equations
                equals (a, a)  =  true          when a≠errorDomain
                equals (a, b)  =  equals (b, a)
                implies (and (equals (a, b), equals (b, c)), equals (a, c)) = true
                          when a≠errorDomain, b≠errorDomain, c≠errorDomain
         end Entries
    exports
        sorts Mapping
        operations
            emptyMap             :  Mapping
            errorMapping         :  Mapping
            update ( _ , _ , _ ) :  Mapping, Domain, Range → Mapping
            apply ( _ , _ )      :  Mapping, Domain → Range
    end exports
    variables
        m          :  Mapping
        d, d1, d2  :  Domain
        r          :  Range
    equations
    [M1]   apply (emptyMap, d)            = errorRange
    [M2]   apply (update (m, d1, r), d2)  = r
                                   when  equals (d1, d2) = true, m≠errorMapping
    [M3]   apply (update (m, d1, r), d2)  = apply (m, d2)
                                   when  equals (d1, d2) = false, r≠errorRange
    end Mappings
```

The operation emptyMap is a constant, and the update operation adds or changes a pair consisting of a domain value and a range value in a mapping. The operation apply returns the range value associated with a domain value or returns errorRange if the mapping has no value for that domain element. Observe the similarity between terms for this specification and the Prolog terms we used in implementing environments and stores earlier in the text. The finite mapping [a |→8,b |→13] corresponds to the term

update (update (emptyMap, a, 8), b, 13),

which represents an object of sort Mapping.

A store structure that associates identifiers represented as strings with natural numbers can be defined in terms of Mappings. The following module instantiates Mappings using the types String and Natural for the domain and range sorts of the mappings, respectively. The operations are renamed to fit the store model of memory.

```
module Stores
    imports Strings, Naturals,
            instantiation of Mappings
                    bind Entries using String for Domain
                                 using Natural for Range
                                 using strEqual for equals
                                 using errorString for errorDomain
                                 using errorNatural for errorRange
                    rename   using Store for Mapping
                             using emptySto for emptyMap
                             using updateSto for update
                             using applySto for apply
end Stores
```

We have introduced enough basic modules to develop an algebraic specification of Wren in section 12.4. However, the notation is sometimes less convenient than desired—for example,

succ (succ (succ (0))) stands for the natural number written as 3 (base-ten)

and

cons (char-a, cons (char-b, nullString)) represents the string literal “ab”.

Additional specification modules can be developed to provide a more conventional notation (see Chapter 6 of [Bergstra89]), but these notational issues are beyond the scope of this book. We also ignore the problem of conflict resolution when several imported modules extend some previously defined module by defining operations with the same name in different ways. Since this topic is dealt with in the references, our presentation will concentrate on semantic rather than syntactic issues. We take a brief look at the mathematical foundations of algebraic semantics in the next section.

Exercises

1. Give the equation(s) for a direct definition of eq? in the module Booleans.

2. Show how the output of tuples can be eliminated by introducing new sorts. Develop a specific example to illustrate the technique.

3. Add function symbols lesseq? and greatereq? to the module Naturals and provide equations to specify their behavior.

4. Add a function symbol exp representing the exponentiation operation to Naturals and provide appropriate equations to specify its behavior.

5. Extend the module for Naturals to a module for Integers by introducing a predecessor operator.

6. Consider the module Mappings. No equations are provided to specify the sort Mapping, so two mappings are equal if they are represented by the same term. Does this notion of equality agree with the normal meaning of equal mappings? What equation(s) can be added to the module to remedy this problem?

7. Define a module that specifies binary trees with natural numbers at its leaf nodes only. Include operations for constructing trees, selecting parts from a tree, and several operations that compute values associated with binary trees, such as "sum of the values at the leaf nodes" and "height of a tree".

8. Redo exercise 7 for binary trees with natural numbers at interior nodes in addition to the leaf nodes.

9. Consider the signature defined by the module Mixtures. List five terms of sort Mixture. Suggest some equations that we may want this specification to satisfy. *Hint*: Consider algebraic properties of the binary operation.

```
module Mixtures
   exports
      sorts Mixture
      operations
         flour        :  Mixture
         sugar        :  Mixture
         salt         :  Mixture
         mix ( _ , _ ) :  Mixture, Mixture → Mixture
   end exports
end Mixtures
```

12.2 MATHEMATICAL FOUNDATIONS

Some very simple modules serve to illustrate the mathematical foundations of algebraic semantics. We simplify the module Booleans to include only the constants false and true and the function not. In a similar way, we limit Naturals to the constant 0, the constructor succ, and the function symbol add. By limiting the operations in this way, we avoid the need for error values in the sorts. See the references, particularly [Ehrig85], for a description of error handling in algebraic specifications.

```
module Bools
    exports
        sorts Boolean
        operations
            true       :  Boolean
            false      :  Boolean
            not ( _ )  :  Boolean → Boolean
    end exports
    equations
    [B1]    not (true)   = false
    [B2]    not (false)  = true
end Bools

module Nats
    imports Bools
    exports
        sorts Natural
        operations
            0                :  Natural
            succ ( _ )       :  Natural → Natural
            add ( _ , _ )    :  Natural, Natural → Natural
    end exports
    variables
        m, n : Natural
    equations
    [N1]    add (m, 0)           =  m
    [N2]    add (m, succ (n))    =  succ (add (m, n))
end Nats
```

Ground Terms

In the previous section we pointed out function symbols that act as constructors provide a way of building terms that represent the objects being defined by a specification. Actually, all function symbols can be used to construct terms that stand for the objects of the various sorts in the signature, although one sort is usually distinguished as the type of interest. We are particularly interested in those terms that have no variables.

Definition : For a given signature Σ = <Sorts,Operations>, the set of **ground terms** T_Σ for a sort S is defined inductively as follows:

1. All constants (nullary function symbols) of sort S in Operations are ground terms of sort S.
2. For every function symbol f : $S_1,\ldots,S_n \rightarrow S$ in Operations, if $t_1,\ldots,t_n$ are ground terms of sorts $S_1,\ldots,S_n$, respectively, then $f(t_1,\ldots,t_n)$ is a ground term of sort S where $S_1,\ldots,S_n,S \in$ Sorts. ∎

Example : The ground terms of sort Boolean for the Bools module consist of all those expressions that can be built using the constants true and false and the operation symbol not. This set of ground terms is infinite.

true, not(true), not(not(true)), not(not(not(true))), ...
false, not(false), not(not(false)), not(not(not(false))), …. ∎

Example : The ground terms of sort Natural in the Nats module are more complex, since two constructors build new terms from old; the patterns are suggested below:

0,	add(0,0),	
succ(0),	add(0,succ(0)),	add(succ(0),0),
succ(succ(0)),	add(0,succ(succ(0))),	add(succ(succ(0)),0),
	add(succ(0),succ(0)),	
succ(succ(succ(0))),	add(0,succ(succ(succ(0)))),	add(succ(succ(succ(0))),0),
	add(succ(0),succ(succ(0))),	add(succ(succ(0)),succ(0)),
:	:	:

∎

If we ignore the equations in these two modules for now, the ground terms must be mutually distinct. On the basis of the signature only (no equations), we have no reason to conclude that not(true) is the same as false and that add(succ(0),succ(0)) is the same as succ(succ(0)).

Σ-Algebras

Algebraic specifications deal only with the syntax of data objects and their operations. Semantics is provided by defining algebras that serve as models of the specifications. **Homogeneous algebras** can be thought of as a single set, called the **carrier**, on which several operations may be defined—for example, the integers with addition and multiplication form a ring. Computer science applications and some mathematical systems, such as vector spaces, need structures with several types. **Heterogeneous** or **many-sorted algebras** have a number of operations that act on a collection of sets. Specifications are modeled by Σ-algebras, which are many-sorted.

Definition: For a given signature Σ, an algebra A is a **Σ-algebra** under the following circumstances:

- There is a one-to-one correspondence between the carrier sets of A and the sorts of Σ.
- There is a one-to-one correspondence between the constants and functions of A and the operation symbols of Σ so that those constants and functions are of the appropriate sorts and functionalities. ∎

A Σ-algebra contains a set for each sort in S and an actual function for each of the function symbols in Σ. For example, let Σ = <Sorts, Operations> be a signature where Sorts is a set of sort names and Operations is a set of function symbols of the form $f : S_1,\ldots,S_n \to S$ where S and each S_i are sort names from Sorts. Then a Σ-algebra A consists of the following:

1. A collection of sets $\{ S_A \mid S \in \text{Sorts} \}$, called the **carrier sets**.
2. A collection of functions $\{ f_A \mid f \in \text{Operations} \}$ with the functionality

   $$f_A : (S_1)_A,\ldots,(S_n)_A \to S_A$$

   for each $f : S_1,\ldots,S_n \to S$ in Operations.

Σ-algebras are called heterogeneous or many-sorted algebras because they may contain objects of more than one sort.

Definition: The **term algebra T_Σ** for a signature Σ = <Sorts, Operations> is constructed as follows. The carrier sets $\{ S_{T_\Sigma} \mid S \in \text{Sorts} \}$ are defined inductively.

1. For each constant c of sort S in Σ we have a corresponding constant "c" in S_{T_Σ}.
2. For each function symbol $f : S_1,\ldots,S_n \to S$ in Σ and any n elements $t_1 \in (S_1)_{T_\Sigma}, \ldots, t_n \in (S_n)_{T_\Sigma}$, the term "$f(t_1,\ldots,t_n)$" belongs to the carrier set $(S)_{T_\Sigma}$.

The functions in the term algebra, corresponding to the function symbols in Operations, are defined by simply forming the literal term that results from applying the function symbol to terms. For each function symbol $f : S_1,\ldots,S_n \to S$ in Σ and any n elements $t_1 \in (S_1)_{T_\Sigma}, \ldots, t_n \in (S_n)_{T_\Sigma}$, we define the function f_{T_Σ} by $f_{T_\Sigma}(t_1, \ldots, t_n)$ = "$f(t_1, \ldots, t_n)$". ∎

The elements of the carrier sets of T_Σ consist of strings of symbols chosen from a set containing the constants and function symbols of Σ together with the special symbols "(", ")", and ",". For example, the carrier set for the term algebra T_Σ constructed from the module Bools contains all the ground terms from the signature, including

"true", "not(true)", "not(not(true))", ...
"false", "not(false)", "not(not(false))", ...

Furthermore, the function not_{T_Σ} maps "true" to "not(true)", which is mapped to "not(not(true))", and so forth.

This term algebra clearly does not specify the intended meaning of Bools since the carrier set is infinite. Also, "false" ≠ "not(true)", which is different from our understanding of the *not* function in Boolean logic. So far we have not accounted for the equations in a specification and what properties they enforce in an algebra.

Definition : For a signature Σ and a Σ-algebra A, the **evaluation function** $eval_A : T_\Sigma \rightarrow A$ from ground terms to values in A is defined as:

$eval_A$ ("c") = c_A for constants c and

$eval_A(\text{"}f(t_1,..,t_n)\text{"}) = f_A(eval_A(t_1),..,eval_A(t_n))$
where each term t_i is of sort S_i for $f : S_1,\ldots,S_m \rightarrow S$ in Operations. ▮

For any Σ-algebra A, the evaluation function from T_Σ must always exist and have the property that it maps function symbols to actual functions in A in a conformal way to be defined later. The term algebra T_Σ is a symbolic algebra—concrete but symbolic.

A Congruence from the Equations

As the function symbols and constants create a set of ground terms, the equations of a specification generate a congruence ≡ on the ground terms. A congruence is an equivalence relation with an additional "substitution" property.

Definition : Let Spec = <Σ,E> be a specification with signature Σ and equations E. The **congruence $\equiv_E$ determined by E on T_Σ** is the smallest relation satisfying the following properties:

1. Variable Assignment: Given an equation lhs = rhs in E that contains variables $v_1,..,v_n$ and given any ground terms $t_1,..,t_n$ from T_Σ of the same sorts as the respective variables,

$$lhs[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n] \equiv_E rhs[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$$

where $v_i \mapsto t_i$ indicates substituting the ground term t_i for the variable v_i. If the equation is conditional, the condition must be valid after the variable assignment is carried out on the condition.

2. Reflexive: For every ground term $t \in T_\Sigma$, $t \equiv_E t$.

3. Symmetric: For any ground terms $t_1, t_2 \in T_\Sigma$, $t_1 \equiv_E t_2$ implies $t_2 \equiv_E t_1$.

4. Transitive: For any terms $t_1, t_2, t_3 \in T_\Sigma$,
   $(t_1 \equiv_E t_2$ and $t_2 \equiv_E t_3)$ implies $t_1 \equiv_E t_3$.

5. Substitution Property: If $t_1 \equiv_E t_1', \ldots, t_n \equiv_E t_n'$ and $f : S_1, \ldots, S_n \rightarrow S$ is any function symbol in Σ, then $f(t_1, \ldots, t_n) \equiv_E f(t_1', \ldots, t_n')$. ∎

Normally, we omit the subscript E and rely on the context to determine which equations apply. To generate an equivalence relation from a set of equations, we take every ground instance of all the equations as a basis, and allow any derivation using the reflexive, symmetric, and transitive properties and the rule that each function symbol preserves equivalence when building ground terms.

For the Bools module, all ground terms are congruent to true or to false.

true ≡ not(false) ≡ not(not(true)) ≡ not(not(not(false))) ≡ ....
false ≡ not(true) ≡ not(not(false)) ≡ not(not(not(true))) ≡ ....

These congruences are easy to prove since no variables are involved. For the Nats module, all ground terms are congruent to one of 0, succ(0), succ(succ(0)), succ(succ(succ(0))), and so forth. For example, the following four terms are congruent:

succ(succ(0)) ≡ add(0,succ(succ(0))) ≡
add(succ(succ(0)),0) ≡ add(succ(0),succ(0)).

We show the proof for add(succ(0),succ(0)) ≡ succ(succ(0)).

add(succ(0),succ(0))
≡ succ(add(succ(0),0)) using [N2] from Nats and a variable assignment with [m |→ succ(0), n |→ 0]
≡ succ(succ(0)) using [N1] from Nats and a variable assignment with [m |→ succ(0)].

Definition : If Spec is a specification with signature Σ and equations E, a Σ-algebra A is a **model** of Spec if for all ground terms t_1 and t_2, $t_1 \equiv_E t_2$ implies $eval_A(t_1) = eval_A(t_2)$. ∎

Example : Consider the algebra A = <{off, on}, {off, on, switch}>, where off and on are constants and switch is defined by

switch(off) = on and switch(on) = off.

Then if Σ is the signature of Bools, A is a Σ-algebra that models the specification defined by Bools.

Boolean$_A$ = {off, on} is the carrier set corresponding to sort Boolean.

Operation symbols of Σ	**Constants/functions of A**
true : Boolean	true$_A$ = on : Boolean$_A$
false : Boolean	false$_A$ = off : Boolean$_A$
not : Boolean $\to$ Boolean	not$_A$ = switch : Boolean$_A$ $\to$ Boolean$_A$

For example, not(true) ≡ false and in the algebra A,

eval$_A$(not(true)) = not$_A$(eval$_A$(true)) = not$_A$(true$_A$) = switch(on) = off, and
eval$_A$(false) = off. ∎

There may be many models for Spec. We now construct a particular Σ-algebra, called the **initial algebra**, that is guaranteed to exist. We take this initial algebra *to be the meaning* of the specification Spec.

The Quotient Algebra

The term algebra T_Σ serves as the starting point in constructing an initial algebra. We build the quotient algebra Q from the term algebra T_Σ of a specification <Σ,E> by factoring out congruences.

Definition : Let <Σ,E> be a specification with Σ = <Sorts, Operations>. If t is a term in T_Σ, we represent its congruence class as [t] = { t' | t $\equiv_E$ t' }. So [t] = [t'] if and only if t $\equiv_E$ t'. These congruence classes form the members of the carrier sets { S_{T_Σ} | S∈Sorts } of the **quotient algebra**, one set for each sort S in the signature. We translate a constant c into the congruence class [c]. The functions in the term algebra define functions in the quotient algebra in the following way:

Given a function symbol f : $S_1,\ldots,S_n \to S$ in Σ, $f_Q([t_1],\ldots,[t_n]) = [f(t_1,..,t_n)]$ for any terms $t_i : S_i$, with $1 \le i \le n$, from the appropriate carrier sets.

The function f_Q is well-defined, since $t_1 \equiv_E t_1'$, …, $t_n \equiv_E t_n'$ implies $f_Q(t_1,..,t_n) \equiv_E f_Q(t_1',..,t_n')$ by the Substitution Property for congruences. ∎

Consider the term algebra for Bools. There are two congruence classes, which we may as well call [true] and [false]. From our previous observation of the congruence of ground terms, we know that the congruence class [true] contains

“true”, “not(false)”, “not(not(true))”, “not(not(not(false)))”, ...

and the congruence class [false] contains
“false”, “not(true)”, “not(not(false))”, “not(not(not(true)))”, ...

The function not_Q is defined in the following way:

not_Q([false]) = [not(false)] = [true], and
not_Q([true]) = [not(true)] = [false].

So the quotient algebra has the carrier set { [true], [false] } and the function not_Q. This quotient algebra is, in fact, an initial algebra for Bools. Initial algebras are not necessarily unique. For example, the algebra

A = <{off, on}, {off, on, switch}>

is also an initial algebra for Bools.

An initial algebra is "finest-grained" in the sense that it equates only those terms required to be equated, and, therefore, its carrier sets contain as many elements as possible. Using the procedure outlined above for developing the term algebra and then the quotient algebra, we can always guarantee that at least one initial algebra exists for any specification.

Homomorphisms

Functions between Σ-algebras that preserve the operations are called Σ-homomorphisms. See Chapter 9 for another description of homomorphisms. These functions are used to compare and contrast algebras that act as models of specifications.

Definition : Suppose that A and B are Σ-algebras for a given signature Σ = <Sorts, Operations>. Then h is a **Σ-homomorphism** if it maps the carrier sets of A to the carrier sets of B and the constants and functions of A to the constants and functions of B, so that the behavior of the constants and functions is preserved. In other words, h consists of a collection $\{ h_S \mid S \in \text{Sorts} \}$ of functions $h_S : S_A \to S_B$ for $S \in$ Sorts such that

$h_S(c_A) = c_B$ for each constant symbol c : S, and

$h_S(f_A(a_1,\ldots,a_n)) = f_B(h_{S_1}(a_1),\ldots,h_{S_n}(a_n))$ for each function symbol

$f : S_1,\ldots,S_n \to S$ in S and any n elements $a_1 \in (S_1)_A,\ldots,a_n \in (S_n)_A$. ∎

If there is a Σ-homomorphism h from A to B and the inverse of h is a Σ-homomorphism from B to A, then h is an **isomorphism** and—apart from renaming carrier sets, constants, and functions—the two algebras are exactly the same.

The notion of Σ-homomorphism is used to define the concept of initial algebra formally.

Definition : A Σ-algebra I in the class of all Σ-algebras that serve as models of a specification with signature Σ is called **initial** if for any Σ-algebra A in the class, there is a unique homomorphism $h : I \to A$. ▮

The quotient algebra Q for a specification is an initial algebra. Therefore for any Σ-algebra A that acts as a model of the specification, there is a unique Σ-homomorphism from Q to A. The function $eval_A : T_\Sigma \to A$ induces the Σ-homomorphism h from Q to A using the definition:

$$h([t]) = eval_A(t) \text{ for each } t \in T_S.$$

The homomorphism h is well defined because if $t_1 \equiv t_2$, $h([t_1]) = eval_A(t_1) = eval_A(t_2) = h([t_2])$.

Any algebra isomorphic to Q is also an initial algebra. So since the quotient algebra Q and the algebra A = <{off, on}, {off, on, switch}> are isomorphic, A is also an initial algebra for Bools. We can now formally define the terms junk and confusion, introduced earlier in this chapter.

Definition : Let <Σ,E> be a specification, let Q be the quotient algebra for <Σ,E>, and let B be an arbitrary model of the specification.

1. If the homomorphism from Q to a Σ-algebra B is not onto (not surjective), then B contains **junk** since B contains values that do not correspond to any terms constructed from the signature.
2. If the homomorphism from Q to B is not one-to-one (not injective), then B exhibits **confusion** since two different values in the quotient algebra correspond to the same value in B. ▮

Consider the quotient algebra for Nats with the infinite carrier set [0], [succ(0)], [succ(succ(0))], and so on. Suppose that we have a 16-bit computer for which the integers consist of the following set of values:

{ -32768, -32767, ..., -1, 0, 1, 2, 3, ..., 32766, 32767 }.

The negative integers are junk with respect to Nats since they cannot be images of any of the natural numbers. On the other hand, all positive integers above 32767 must be confusion. When mapping an infinite carrier set onto a finite machine, confusion must occur.

Consistency and Completeness

Consistency and completeness are two issues related to junk and confusion. The following examples illustrate these notions. Suppose we want to add a predecessor operation to naturals by importing Naturals (the original version) and defining a predecessor function pred.

```
module Predecessor1
    imports Booleans, Naturals
    exports
        operations
            pred ( _ ) : Natural → Natural
    end exports
    variables
        n : Natural
    equations
    [P1]    pred (succ (n)) = n
end Predecessor1
```

We say that Naturals is a subspecification of Predecessor$_1$ since the signature and equations of Predecessor$_1$ include the signature and equations of Naturals. We have added a new congruence class [pred(0)], which is not congruent to 0 or any of the successors of 0. We say that [pred(0)] is junk and that Predecessor$_1$ is not a **complete extension** of Naturals. We can resolve this problem by adding the equation [P2] pred(0) = 0 (or [P2] pred(0) = errorNatural).

Suppose that we define another predecessor module in the following way:

```
module Predecessor2
    imports Booleans, Naturals
    exports
        operations
            pred ( _ ) : Natural → Natural
    end exports
    variables
        n  : Natural
    equations
    [P1]    pred (n)  =  sub (n, succ (0))
    [P2]    pred (0)  =  0
end Predecessor2
```

The first equation specifies the predecessor by subtracting one, and the second equation is carried over from the "fix" for Predecessor$_1$. In the module Naturals, we have the congruence classes:

[errorNatural], [0], [succ(0)], [succ(succ(0))], ...

With the new module Predecessor$_2$, we have pred(0) = sub(0,succ(0)) = errorNatural by [P1] and [N5], and pred(0) = 0 by [P2]. So we have reduced the number of congruence classes, since [0] = [errorNatural]. Because this has introduced confusion, we say that Predecessor_2 is not a **consistent extension** of Naturals.

Definition : Let Spec be a specification with signature Σ = <Sorts, Operations> and equations E. Suppose SubSpec is a subspecification of Spec with sorts SubSorts (a subset of Sorts) and equations SubE (a subset of E). Let T and SubT represent the terms of Sorts and SubSorts, respectively.

- Spec is a **complete extension** of SubSpec if for every sort S in SubSorts and every term t_1 in T, there exists a term t_2 in SubT such that t_1 and t_2 are congruent with respect to E.
- Spec is a **consistent extension** of SubSpec if for every sort S in SubSorts and all terms t_1 and t_2 in T, t_1 and t_2 are congruent with respect to E if and only if t_1 and t_2 are congruent with respect to SubE. ∎

Exercises

1. Describe an initial algebra for the simplified Nats module given in this section.

2. Use the specification of Booleans in section 12.1 to prove the following congruences:

   a) and(not(false),not(true)) ≡ false

   b) or(not(false),not(true)) ≡ true

3. Use the specification of Naturals in section 12.1 to prove the following congruences:

   a) sub (10, succ (succ (succ (succ (succ (succ(0)))))))
      ≡ succ (succ (succ (succ (0))))

   b) mul (succ (succ (0)), succ (succ (0))) ≡ succ (succ (succ (succ (0))))

   c) less? (succ (0), succ (succ (succ (0)))) ≡ true

4. Each of the following algebras are Σ-algebras for the signature of Nats. Identify those that are initial and define the homomorphisms from the initial algebras to the other algebras. Do any of these algebras contain confusion or junk?

   a) A = <{ 0, 1, 2, 3, … }, $\{0_A, succ_A, add_A\}$> where $0_A = 0$, $succ_A = \lambda n\ .\ n+1$, and $add_A = \lambda m\ .\ \lambda n\ .\ m+n$.

   b) B = <{ 0, 1, 2 }, $\{0_B, succ_B, add_B\}$> where $0_B = 0$, $succ_B(0)= 1$, $succ_B(1)= 2$, $succ_B(2)= 0$, and $add_B = \lambda m\ .\ \lambda n\ .\ m+n$ (modulo 3).

c) C = <{ …, -2, -1, 0, 1, 2, 3, … }, {0_C, $succ_C$, add_C}> where $0_C = 0$, $succ_C = \lambda n . n+1$, and $add_C = \lambda m . \lambda n . m+n$.

d) D = <{ zero, succ(zero), succ(succ(zero)), … }, {0_D, $succ_D$, add_D}> where 0_D = zero, $succ_D = \lambda n$. succ(n), add_D(m,zero) = m, and add_D(m,$succ_D$(n)) = succ(add_D(m,n)).

5. List five different terms in the term algebra T_Σ for the specification of stores in the module at the end of section 12.1. Describe the quotient algebra for Mappings, including two additional equations:

[M4] update (update (m, d_1, r_1), d_2, r_2) = update (update (m, d_2, r_2), d_1, r_1) *when* $d_1 \neq d_2$

[M5] update (update (m, d_1, r_1), d_1, r_2) = update (m, d_1, r_2).

6. Consider the following module that defines a specification <Σ,E> with signature Σ and equations E. Ignore the possibility of an errorBoolean value in the sort.

```
module Booleans
    exports
        sorts Boolean
        operations
                true : Boolean
                false : Boolean
                not ( _ ) : Boolean → Boolean
                nand ( _ , _ ) : Boolean, Boolean → Boolean
    end exports
    variables
        b : Boolean
    equations
        [B1]    nand (false, false) = true
        [B2]    nand (false, true) = false
        [B3]    nand (true, false) = false
        [B4]    nand (true, true) = false
        [B5]    not (b) = nand (b, b)
    end Booleans
```

a) Give an induction definition of the carrier set of the term algebra T_Σ for this Σ.

b) Carefully describe the quotient algebra Q for this specification.

c) Describe another algebra A whose carrier set has only *one* element and that models this specification. Define a homomorphism from Q to A.

d) Describe another algebra B whose carrier set has *three* elements and that models this specification. Define a homomorphism from Q to B.

12.3 USING ALGEBRAIC SPECIFICATIONS

Before considering the algebraic semantics for Wren, we take a detour to discuss several other uses of algebraic specifications. Defining abstract data types has proved to be the most productive application of these specification methods so far. In the first part of this section we develop and discuss the specification of abstract data types using algebraic methods. Then in the second part of this section we return to the concept of abstract syntax and see that it can be put on a more formal foundation by exploiting algebraic specifications and their corresponding algebras.

Data Abstraction

The main problem in creating large software systems is that their complexity can exceed the programmers' powers of comprehension. Using abstraction provides a fundamental technique for dealing with this complexity. Abstraction means that a programmer concentrates on the essential features of the problem while ignoring details and characteristics of concrete realizations in order to moderate the magnitude of the complexity.

Abstraction aids in the constructing, understanding, and maintaining of systems by reducing the number of details a programmer needs to understand while working on one part of the problem. The reliability of a system is enhanced by designing it in modules that maintain a consistent level of abstraction, and by permitting only certain operations at each level of abstraction. Any operation that violates the logical view of the current level of abstraction is prohibited. A programmer uses procedural and data abstractions without knowing how they are implemented (called **information hiding**). Unnecessary details of data representation or of an operation's implementation are hidden from those who have no need to see them.

Data abstraction refers to facilities that allow the definition of new sorts of data and operations on that data. Once the data and operations have been defined, the programmer forgets about the implementation and simply deals with their logical properties. The goal of data abstraction is to separate the logical properties of the data and operations from the implementation. Programmers work with abstract data types when they use the predefined types in a programming language. The objects and operations of integer type are used in a program based solely on their logical characteristics; programmers need know nothing of the representation of integers or of the implementation of the arithmetic operations. This information hiding allows them to consider problems at a higher level of abstraction by ignoring implementation details. High-level strategies should not be based on low-level details.

The full power of abstraction becomes evident only when programmers can create abstract data types for themselves. Many modern programming languages provide support for the specification of ADTs. Three facilities are desirable in a programming language for the creation of ADTs:

1. **Information Hiding** : The compiler should ensure that the user of an ADT does not have access to the representation (of the values) and implementation (of the operations) of an ADT.
2. **Encapsulation** : All aspects of the specification and implementation of an ADT should be contained in one or two syntactic unit(s) with a well-defined interface to the users of the ADT. The Ada package, the Modula-2 module, and the class feature in object-oriented programming languages are examples of encapsulation mechanisms.
3. **Generic types** (parameterized modules): There should be a way of defining an ADT as a template without specifying the nature of all its components. Such a generic type will be instantiated when the properties of its missing component values are instantiated.

Instead of delving into the definition of ADTs in programming languages, we return now to a more formal discussion of data abstraction in the context of algebraic specification as already examined in the first two sections of this chapter.

A Module for Unbounded Queues

We start by giving the signature of a specification of queues of natural numbers.

```
module Queues
    imports Booleans, Naturals
    exports
        sorts Queue
        operations
            newQ            :  Queue
            errorQueue      :  Queue
            addQ ( _ , _ )  :  Queue, Natural  → Queue
            deleteQ ( _ )   :  Queue           → Queue
            frontQ ( _ )    :  Queue           → Natural
            isEmptyQ ( _ ):    Queue           → Boolean
    end exports
end Queues
```

Given only the signature of Queues, we have no justification for assuming any properties of the operations other than their basic syntax. Except for the names of the operations, which are only meaningless symbols at this point, this module could be specifying stacks instead of queues. One answer to this ambiguity is to define the characteristic properties of the queue ADT by describing informally what each operation does—for example:

- The function isEmptyQ(q) returns true if and only if the queue q is empty.
- The function frontQ(q) returns the natural number in the queue that was added earliest without being deleted yet.
- If q is an empty queue, frontQ(q) is an error value.

Several problems arise with this sort of informal approach. The descriptions are ambiguous, depending on terms that have not been defined—for example, "empty" and "earliest". The properties depend heavily on the names used for the operations and what they suggest. The names will be of no use with a completely new data type. On the other hand, a programmer may be tempted to define the meaning of the operations in terms of an implementation of them, say as an array with two index values identifying the front and rear of the queue, but this defeats the whole intent of data abstraction, which is to separate logical properties of data objects from their concrete realization.

A more formal approach to specifying the properties of an ADT is through a set of axioms in the form of module equations that relate the operations to each other. We insert the following sections into the module Queues:

```
variables
    q  :  Queue
    m  :  Natural

equations
[Q1]   isEmptyQ (newQ)          = true
[Q2]   isEmptyQ (addQ (q,m))    = false     when q≠errorQueue, m≠errorNatural
[Q3]   delete (newQ)            = newQ
[Q4]   deleteQ (addQ (q,m))     = if ( isEmptyQ(q), newQ, addQ(deleteQ(q),m))
                                            when m≠errorNatural
[Q5]   frontQ (newQ)            = errorNatural
[Q6]   frontQ (addQ (q,m))      = if ( isEmptyQ(q), m, frontQ(q) )
                                            when m≠errorNatural
```

The decision to have delete(newQ) return newQ is arbitrary. Some other time we might want delete(newQ) to be errorQueue when describing the behavior of a queue.

Implementing Queues as Unbounded Arrays

Assuming that the axioms correctly specify the concept of a queue, they can be used to verify that an implementation is correct. A realization of an abstract data type will consist of a representation of the objects of the type, implementations of the operations, and a **representation function** Φ that maps terms in the model onto the abstract objects in such a way that the axioms are satisfied. For example, say we want to represent queues as arrays with two pointers, one to the front of the queue and one to the rear. Note that the implementation is simplified by defining unbounded arrays, since the queues that have been described are unbounded.

To enhance the readability of this presentation, we use abbreviations such as "m=n" for eq?(m,n) and "m≤n" for not(greater?(m,n)) from now on. The notion of an unbounded array is presented as an abstract data type in the following module:

```
module Arrays
    imports Booleans, Naturals
    exports
        sorts Array
        operations
            newArray   : Array
            errorArray : Array
            assign ( _ , _ , _ ) : Array, Natural, Natural → Array
            access ( _ , _ )     : Array, Natural → Natural
    end exports
    variables
        arr     :  Array
        i, j, m :  Natural
    equations
    [A1]    access (newArray, i) = errorNatural
    [A2]    access (assign (arr, i, m), j) = if ( i = j, m, access(arr,j) )
                                                  when m≠errorNatural
end Arrays
```

The implementation of the ADT Queue using the ADT Array has the following set of triples as its objects:

ArrayQ = { <arr,f,e> | arr : Array and f,e : Natural and f≤e }.

The operations over ArrayQ are defined as follows:

```
[AQ1]    newAQ = <newArray,0,0>
[AQ2]    addAQ (<arr,f,e>, m)     = <assign(arr,e,m),f,e+1>
```

[AQ3]	deleteAQ (<arr,f,e>)	= *if* (f = e, <newArray,0,0>, <arr,f+1,e>)
[AQ4]	frontAQ (<arr,f,e>)	= *if* (f = e, errorNatural, access(arr,f))
[AQ5]	isEmptyAQ (<arr,f,e>)	= (f = e) *when* arr≠errorArray

The array queues are related to the abstract queues by a homomorphism, called a representation function,

$$\Phi : \{ \text{ArrayQ,Natural,Boolean} \} \rightarrow \{ \text{Queue,Natural,Boolean} \},$$

defined on the objects and operations of the sort. We use the symbolic terms "Φ(arr,f,e)" to represent the abstract queue objects in Queue.

For arr : Array, m : Natural, and b : Boolean,

Φ (<arr,f,e>) = Φ(arr,f,e) *when* f≤e

Φ (<arr,f,e>) = errorQueue *when* f>e

Φ (m) = m

Φ (b) = b

Φ (newAQ) = newQ

Φ (addAQ) = addQ

Φ (deleteAQ) = deleteQ

Φ (frontAQ) = frontQ

Φ (isEmptyAQ) = isEmptyQ

Under the homomorphism, the five equations that define operations for the array queues map into five equations describing properties of the abstract queues.

[D1]	newQ	= Φ(newArray,0,0)
[D2]	addQ (Φ(arr,f,e), m)	= Φ(assign(arr,e,m),f,e+1)
[D3]	deleteQ (Φ(arr,f,e))	= *if* (f = e, Φ(newArray,0,0), Φ(arr,f+1,e))
[D4]	frontQ (Φ(arr,f,e))	= *if* (f = e, errorNatural, access(arr,f))
[D5]	isEmptyQ (Φ(arr,f,e))	= (f = e)

As an example, consider the image of [AQ2] under Φ.

Assume [AQ2] addAQ (<arr,f,e>,m) = <assign (arr,e,m),f,e+1>.

Then addQ (Φ(arr,f,e),m) = Φ(addAQ) (Φ(<arr,f,e>),Φ(m)>)
= Φ(addAQ (<arr,f,e>,m))
= Φ(assign(arr,e,m),f,e+1),

which is [D2].

The implementation is correct if its objects can be shown to satisfy the queue axioms [Q1] to [Q6] for arbitrary queues of the form q = Φ(arr,f,e) with f≤e and arbitrary elements m of Natural, given the definitions [D1] to [D5] and the equations for arrays. First we need a short lemma.

Lemma: For any queue Φ(a,f,e) constructed using the operations of the implementation, f≤e.

Proof: The only operations that produce queues are newQ, addQ, and deleteQ, the constructors in the signature. The proof is by induction on the number of applications of these operations.

Basis: Since newQ = Φ(newArray,0,0), f≤e.

Induction Step : Suppose that Φ(a,f,e) has been constructed with n applications of the operations and that f≤e.

Consider a queue constructed with one more application of these functions, for a total of n+1.

Case 1: The n+1st operation is addQ.
But addQ (Φ(a,f,e),m) = Φ(assign (a,f,m), f, e+1) has f≤e+1.

Case 2: The n+1st operation is deleteQ.
But deleteQ (Φ(a,f,e)) = *if* (f = e, Φ(arr,f,e), Φ(arr,f+1,e)).
If f=e, then f≤e, and if f<e, then f+1≤e. ∎

The proof of the lemma is an example of a general principle, called **structural induction** because the induction covers all of the ways in which the objects of the data type may be constructed (see the discussion of structural induction in Chapter 8). The goal is to prove a property that holds for all the values of a particular sort, and the induction applies to those operations (the constructors) that produce elements of the sort. For the lemma, the constructors for Queue consist of newQ, addQ, and deleteQ. The general principle can be described as follows:

> **Structural Induction** : Suppose f_1, f_2, …, f_n are the operations that act as constructors for an abstract data type S, and P is a property of values of sort S. If the truth of P for all arguments of sort S for each f_i implies the truth of P for the results of all applications of f_i that satisfy the syntactic specification of S, it follows that P is true of all values of the data type. The basis case results from those constructors with no arguments—namely, the constants of sort S.

To enable the verification of [Q4] as part of proving the validity of this queue implementation, it is necessary to extend Φ for the following values:

For any f : Natural and arr : Array, Φ(arr,f,f) = newQ.

This extension is consistent with definition [D1].

Verification of Queue Axioms

Let q = Φ(a,f,e) be an arbitrary queue and let m be an arbitrary element of Natural.

[Q1] isEmptyQ (newQ) = isEmptyQ (Φ(newArray,0,0)) by [D1]
= (0 = 0) = true by [D5].

[Q2] isEmptyQ (addQ (Φ(arr,f,e),m))
= isEmptyQ (Φ(assign(arr,e,m),f,e+1) by [D2]
= (f = e+1) = false, since f≤e by [D5] and the lemma.

[Q3] deleteQ (newQ) = deleteQ (Φ(newArray,0,0)) by [D1]
= Φ(newArray,0,0) = newQ by [D3] and [D1].

[Q4] deleteQ (addQ (Φ(arr,f,e), m))
= deleteQ (Φ(assign(arr,e,m),f,e+1)) by [D2]
= Φ(assign(arr,e,m),f+1,e+1) by [D3].

Case 1 : f = e, that is, isEmptyQ (Φ(arr,f,e)) = true.
Then Φ(assign(arr,e,m),f+1,e+1) = newQ by [D1].

Case 2 : f < e, that is, isEmptyQ (Φ(arr,f,e)) = false.
Then Φ(assign(arr,e,m),f+1,e+1) = addQ (Φ(arr,f+1,e), m) by [D2]
= addQ (deleteQ (Φ(arr,f,e)), m) by [D3].

[Q5] frontQ (newQ) = frontQ (Φ(newArray,0,0)) by [D1]
= errorNatural since 0 = 0 by [D4].

[Q6] frontQ (addQ (Φ(arr,f,e), m)) = frontQ (Φ(assign(arr,e,m),f,e+1)) by [D2]
= access (assign(arr,e,m), f) by [D4].

Case 1 : f = e, that is, isEmptyQ (Φ(arr,f,e)) = true.
So access (assign(arr,e,m), f) = access (assign (arr,e,m), e) = m by [A2].

Case 2 : f < e, that is, isEmptyQ (Φ(arr,f,e)) = false.
Then access (assign (arr,e,m), f) = access (arr,f)
= frontQ (Φ(arr,f,e)) by [A2] and [D4].

Since the six axioms for the unbounded queue ADT have been verified, we know that the implementation via the unbounded arrays is correct. ∎

ADTs As Algebras

In the previous section we defined Σ-algebras, the many-sorted algebras that correspond to specifications with signature Σ. Now we apply some of the results to the queue ADT. Recall that any signature Σ defines a Σ-algebra T_Σ of all the terms over the signature, and that by taking the quotient algebra Q defined by the congruence based on the equations E of a specification, we get an initial algebra that serves as the finest-grained model of a specification <Σ,E>.

Example : An instance of the queue ADT has operations involving three sorts of objects—namely, Natural, Boolean, and the type being defined, Queue. Some authors designate the type being defined as the **type of interest**. In this context, a graphical notation has been suggested (see Figure 12.1) to define the **signature** of the operations of the algebra.

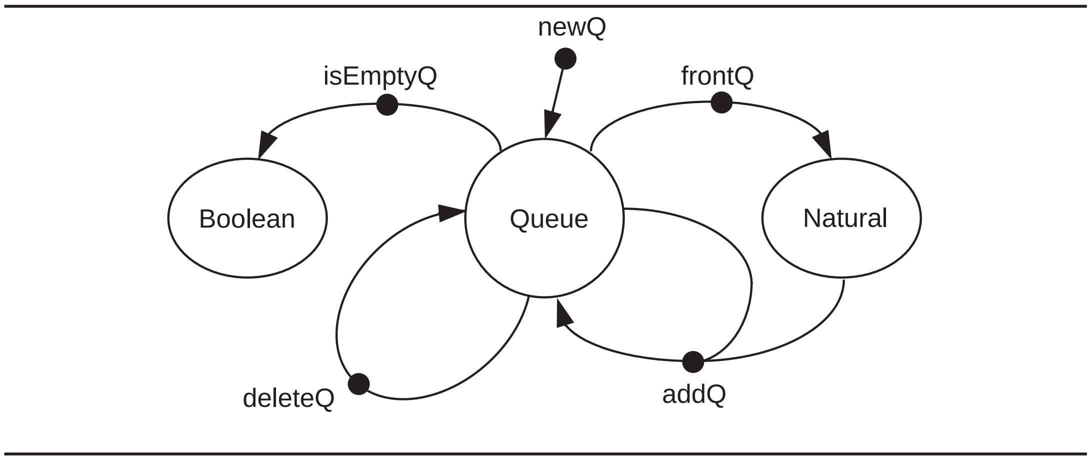

Figure 12.1: Signature of Queues

The signature of the queue ADT defines a term algebra T_Σ, sometimes called a **free word algebra** , formed by taking all legal combinations of operations that produce queue objects. The values in the sort Queue are those produced by the constructor operations. For example, the following terms are elements of T_Σ (we use common abbreviations for natural numbers now, such as 5 for succ(succ(succ(succ(succ(0))))):

newQ,

addQ (newQ,5), and

deleteQ (addQ (addQ (deleteQ (newQ),9),15)).

The term **free** for such an algebra means that the operations are combined in any way satisfying the syntactic constraints, and that all such terms are distinct objects in the algebra. The properties of an ADT are given by a set E of equations or axioms that define identities among the terms of T_Σ.

So the queue ADT is not a free algebra, since the axioms recognize certain terms as being equal. For example:

deleteQ (newQ) = newQ and

deleteQ (addQ (addQ (deleteQ (newQ), 9), 15)) = addQ (newQ, 15).

The equations define a congruence $\equiv_E$ on the free algebra of terms as described in section 12.2. That equivalence relation defines a set of equivalence classes that partition T_Σ.

$$[\ t\ ]_E = \{\ u \in T_\Sigma \mid u \equiv_E t\ \}$$

For example, $[\ \text{newQ}\ ]_E = \{$ newQ, deleteQ(newQ), deleteQ(deleteQ(newQ)), … $\}$.

The operations of the ADT can be defined on these equivalence classes as in the previous section:

> For an n-ary operation $f \in S$ and $t_1,t_2,\ldots,t_n \in T_\Sigma$,
> let $f_Q([t_1],[t_2],\ldots,[t_n]) = [f(t_1,t_2,\ldots,t_n)]$.

The resulting (quotient) algebra, also called $T_{\Sigma,E}$, *is* the abstract data type being defined. When manipulating the objects of the (quotient) algebra $T_{\Sigma,E}$ the normal practice is to use representatives from the equivalence classes.

Definition : A **canonical** or **normal form** for the terms in a quotient algebra is a set of distinct representatives, one from each equivalence class. ∎

Lemma: For the queue ADT $T_{\Sigma,E}$ each term is equivalent to the value newQ or to a term of the form addQ(addQ(…addQ(addQ(newQ,m_1),m_2),…),m_{n-1}),m_n) for some $n \geq 1$ where $m_1,m_2,\ldots,m_n$: Natural.

Proof: The proof is by structural induction.

Basis: The only constant in T_Σ is newQ, which is in normal form.

Induction Step : Consider a queue term t with more than one application of the constructors (newQ, addQ, deleteQ), and assume that any term with fewer applications of the constructors can be put into normal form.

Case 1 : t = addQ(q,m) will be in normal form when q, which has fewer constructors than t, is in normal form.

Case 2 : Consider t = deleteQ(q) where q is in normal form.

> **Subcase a** : q = newQ. Then deleteQ(q) = newQ is in normal form.
>
> **Subcase b** : q = addQ(p,m) where p is in normal form.
>
> Then deleteQ(addQ(p,m)) = *if* (isEmptyQ(p), newQ,addQ(deleteQ(p),m)).
>
> If p is empty, deleteQ(q) = newQ is in normal form.
>
> If p is not empty, deleteQ(q) = addQ(deleteQ(p),m). Since deleteQ(p) has fewer constructors than t, it can be put into normal form, so that deleteQ(q) is in normal form. ∎

A canonical form for an ADT can be thought of as an “abstract implementation” of the type. John Guttag [Guttag78b] calls this a **direct implementation** and represents it graphically as shown in Figure 12.2.

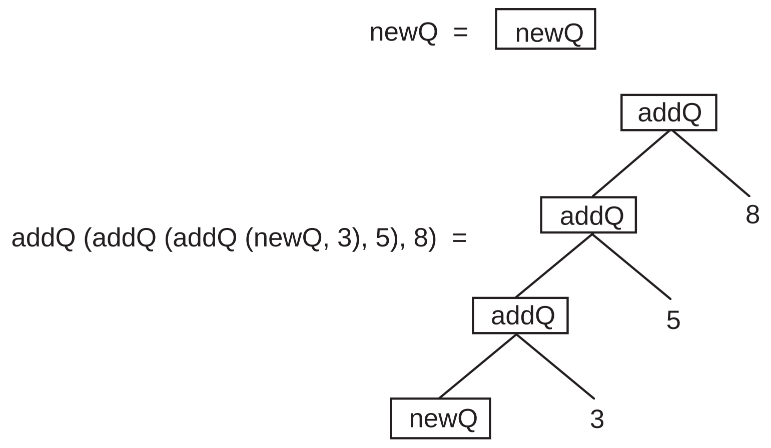

Figure 12.2: Direct Implementation of Queues

The canonical form for an ADT provides an effective tool for proving properties about the type.

Lemma: The representation function Φ that implements queues as arrays is an onto function.

Proof: Since any queue can be written as newQ or as addQ(q,m), we need to handle only these two forms. By [D1], Φ(newArray,0,0) = newQ.

Assume as an induction hypothesis that q = Φ(arr,f,e) for some array. Then by [D2], Φ(assign(arr,e,m),f,e+1) = addQ (Φ(arr,f,e),m).

Therefore any queue is the image of some triple under the representation function Φ. ∎

Given an ADT with signature Σ, operations in Σ that produce an element of the type of interest have already been called **constructors**. Those operations in Σ whose range is an already defined type of "basic" values are called **selectors**. The operations of Σ are partitioned into two disjoint sets, Con the set of constructors and Sel the set of selectors. The selectors for Queues are frontQ and isEmptyQ.

Definition: A set of equations for an ADT is **sufficiently complete** if for each ground term $f(t_1,t_2,…,t_n)$ where $f \in$ Sel, the set of selectors, there is an element u of a predefined type such that $f(t_1,t_2,…,t_n) \equiv_E u$. This condition means there are sufficient axioms to make the derivation to u.

Theorem: The equations in the module Queues are sufficiently complete.

Proof:
1. Every queue can be written in normal form as newQ or as addQ(q,m).

2. isEmptyQ(newQ) = true, isEmptyQ(addQ(q,m)) = false, frontQ(newQ) = errorNatural, and frontQ(addQ(q,m)) = m or frontQ(q) (use induction). ∎

Abstract Syntax and Algebraic Specifications

Throughout the text we have emphasized the importance of abstract syntax in the definition of programming language semantics. In particular, we have stressed several points about abstract syntax:

- In a language definition we need to specify only the meaning of the syntactic forms given by the abstract syntax, since this formalism furnishes all the essential syntactic constructs in the language. Details in the concrete syntax (BNF) may be ignored. No harm arises from an ambiguous abstract syntax since its purpose is not syntactic analysis (parsing). Abstract syntax need only delineate the structure of possible language constructs that can occur in the programs to be analyzed semantically.
- The abstract syntax of a programming language may take many different forms, depending on the semantic techniques that are applied to it. For instance, the abstract syntax for structural operational semantics has little resemblance to that for denotational semantics in its format.

The variety of abstract syntax and its tolerance of ambiguity raises questions concerning the nature of abstract syntax and its relation to the language defined by the concrete syntax. Answers to these questions can be found by analyzing the syntax of programming languages in the context of algebraic specifications.

To illustrate how a grammar can be viewed algebraically, we begin with a small language of integer expressions whose concrete syntax is shown in Figure 12.3.

```
<expr> ::= <term>
<expr> ::= <expr> + <term>
<expr> ::= <expr> – <term>
<term> ::= <element>
<term> ::= <term> * <element>
<element> ::= <identifier>
<element> ::= ( <expr> )
```

Figure 12.3: Concrete Syntax for Expressions

To put this syntactic specification into an algebraic setting, we define a signature Σ that corresponds exactly to the BNF definition. Each nonterminal becomes a sort in Σ, and each production becomes a function symbol whose syntax captures the essence of the production. The signature of the concrete syntax is given in the module Expressions.

```
module Expressions
    exports
        sorts Expression, Term, Element, Identifier
        operations
            expr ( _ )      :  Term               → Expression
            add ( _ , _ ) :  Expression, Term   → Expression
            sub ( _ , _ ) :  Expression, Term   → Expression
            term ( _ )      :  Element            → Term
            mul ( _ , _ ) :  Term, Element      → Term
            elem ( _ )      :  Identifier         → Element
            paren ( _ )     :  Expression         → Element
    end exports
end Expressions
```

Observe that the terminal symbols in the grammar are "forgotten" in the signature since they are embodied in the unique names of the function symbols. Now consider the collection of Σ-algebras following this signature. Since the specification has no equations, the term algebra T_Σ is initial in the collection of all Σ-algebras, meaning that for any Σ-algebra A, there is a unique homomorphism $h : T_\Sigma \rightarrow A$. The elements of T_Σ are terms constructed using the function symbols in Σ. Since this signature has no constants, we assume a set of constants of sort Identifier and represent them as structures of the form ide(x) containing atoms as the identifiers. Think of these structures as the tokens produced by a scanner. The expression "x * (y + z)" corresponds to the following term in T_Σ:

```
t = expr (mul (term (elem (ide(x))),
                    paren (add (expr (term (elem (ide(y)))),
                                    term (elem (ide(z))))))).
```

Constructing such a term corresponds to parsing the expression. In fact, the three algebras, the term algebra T_Σ, the collection of expressions satisfying the BNF definition, and the collection of parse (derivation) trees of expressions are isomorphic. Consider the two trees in Figure 12.4. The one on the left is the derivation tree for "x * (y + z)", and the other one represents its associated term in T_Σ.

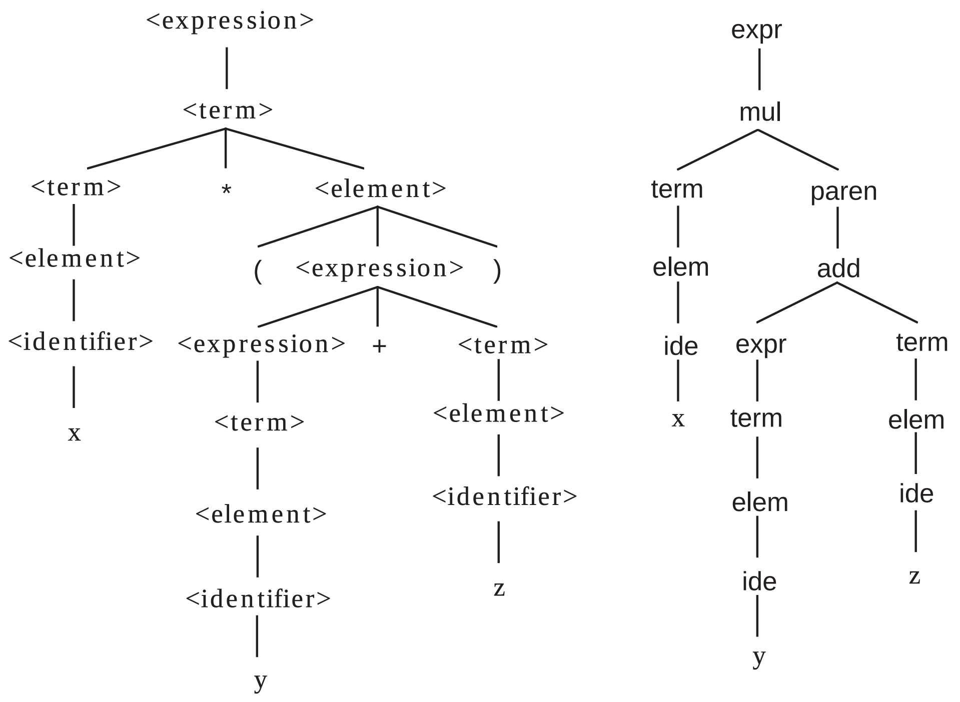

Figure 12.4: Derivation Tree and Algebraic Term

If the concrete syntax of a programming language coincides with the initial term algebra of a specification with signature Σ, what does its abstract syntax correspond to? Consider the following algebraic specification of abstract syntax for the expression language.

```
module AbstractExpressions
    exports
        sorts AbstractExpr, Symbol
        operations
            plus ( _ , _ )  :  AbstractExpr, AbstractExpr  → AbstractExpr
            minus ( _ , _ ) :  AbstractExpr, AbstractExpr  → AbstractExpr
            times ( _ , _ ) :  AbstractExpr, AbstractExpr  → AbstractExpr
            ide ( _ )       :  Symbol                      → AbstractExpr
    end exports
end AbstractExpressions
```

Employing the set Symbol of symbolic atoms used as identifiers in the expression language, we can construct terms with the four constructor function symbols in the AbstractExpressions module to represent the abstract syntax trees for the language. These freely constructed terms form a term algebra, call it A, according to the signature of AbstractExpressions. In addition, A also serves as a model of the specification in the Expressions module; that is, A is a Σ-algebra as evidenced by the following interpretation of the sorts and function symbols:

$\text{Expression}_A = \text{Term}_A = \text{Element}_A = \text{AbstractExpr}$

$\text{Identifier}_A = \{ \text{ide}(x) \mid x : \text{Symbol} \}$.

Operations:

expr_A : AbstractExpr → AbstractExpr
defined by $\text{expr}_A(e) = e$

add_A : AbstractExpr, AbstractExpr → AbstractExpr
defined by $\text{add}_A(e_1,e_2) = \text{plus}(e_1,e_2)$

sub_A : AbstractExpr, AbstractExpr → AbstractExpr
defined by $\text{sub}_A(e_1,e_2) = \text{minus}(e_1,e_2)$

term_A : AbstractExpr → AbstractExpr
defined by $\text{term}_A(e) = e$

mul_A : AbstractExpr, AbstractExpr → AbstractExpr
defined by $\text{mul}_A(e_1,e_2) = \text{times}(e_1,e_2)$

elem_A : Identifier → AbstractExpr
defined by $\text{elem}_A(e) = e$

paren_A : AbstractExpr → AbstractExpr
defined by $\text{paren}_A(e) = e$

Under this interpretation of the symbols in Σ, the term t, shown in Figure 12.4, becomes a value in the Σ-algebra A:

$$\begin{aligned}
t_A &= (\text{expr}(\text{mul}(\text{term}(\text{elem}(\text{ide}(x))),\\
&\qquad \text{paren}(\text{add}(\text{expr}(\text{term}(\text{elem}(\text{ide}(y)))), \text{term}(\text{elem}(\text{ide}(z)))))))))_A\\
&= \text{expr}_A(\text{mul}_A(\text{term}_A(\text{elem}_A(\text{ide}(x))),\\
&\qquad \text{paren}_A(\text{add}_A(\text{expr}_A(\text{term}_A(\text{elem}_A(\text{ide}(y)))), \text{term}_A(\text{elem}_A(\text{ide}(z)))))))\\
&= \text{expr}_A(\text{mul}_A(\text{term}_A(\text{ide}(x)),\\
&\qquad \text{paren}_A(\text{add}_A(\text{expr}_A(\text{term}_A(\text{ide}(y))), \text{term}_A(\text{ide}(z))))))\\
&= \text{expr}_A(\text{mul}_A(\text{ide}(x), \text{paren}_A(\text{add}_A(\text{expr}_A(\text{ide}(y)), \text{ide}(z)))))\\
&= \text{mul}_A(\text{ide}(x), \text{add}_A(\text{ide}(y), \text{ide}(z)))\\
&= \text{times}(\text{ide}(x), \text{plus}(\text{ide}(y), \text{ide}(z)))
\end{aligned}$$

The last term in this evaluation represents the abstract syntax tree in A that corresponds to the original expression “x * (y + z)”.

Each version of abstract syntax is a Σ-algebra for the signature associated with the grammar that forms the concrete syntax of the language. Furthermore, any Σ-algebra serving as an abstract syntax is a homomorphic image of T_Σ, the initial algebra for the specification with signature Σ. Generally, Σ-algebras acting as abstract syntax will contain confusion; the homomorphism from T_Σ will not be one-to-one. This confusion reflects the abstracting process: By confusing elements in the algebra, we are suppressing details in the syntax. The expressions "x+y" and "(x+y)", although distinct in the concrete syntax and in T_Σ, are indistinguishable when mapped to plus(ide(x),ide(y)) in A.

Any Σ-algebra for the signature resulting from the concrete syntax can serve as the abstract syntax for some semantic specification of the language, but many such algebras will be so confused that the associated semantics will be trivial or absurd. The task of the semanticist is to choose an appropriate Σ-algebra that captures the organization of the language in such a way that appropriate semantics can be attributed to it.

Exercises

1. Define suitable canonical forms for the following ADTs, and prove their correctness:

   a) Unbounded Array

   b) Stack of natural numbers

2. Define a parameterized module for queues in which the items in the queues are unspecified until the module is instantiated. Give an instantiation of the module.

3. Define a module that specifies the integers including operations for successor, predecessor, addition, equality, and less than. Determine the form of canonical terms for the (initial) quotient algebra for the specification, proving that the forms chosen are adequate.

4. Determine the form of canonical terms for the (initial) quotient algebra generated by the following module that specifies lists of natural numbers. Prove that the canonical forms are sufficient and argue that the choice is minimal.

```
module NatLists
   imports Booleans, Naturals
   exports
   sorts List
   functions
        emptyList      :  List
        mkList ( _ )   :  Natural → List
        concat ( _ , _ ) :  List, List → List
        consL ( _ , _ )  :  Natural, List → List
        consR ( _ , _ )  :  List, Natural → List
        empty? ( _ )     :  List → Boolean
        length ( _ )     :  List → Natural
    end exports

    variables
        s, s1, s2, s3  :  List
        m              :  Natural

    equations
    [NL1]  concat (s, emptyList)          =  s
    [NL2]  concat (emptyList, s)          =  s
    [NL3]  concat (concat (s1, s2), s3)   =  concat (s1, concat (s2, s3))
    [NL4]  consL (m, s)                   =  concat (mkList (m), s)
    [NL5]  consR (s, m)                   =  concat (s, mkList (m))
    [NL6]  empty? (emptyList)             =  true
    [NL7]  empty? (mkList (m))            =  false        when m ≠ errorNatural
    [NL8]  empty? (concat (s1, s2))       =  and (empty? (s1), empty? (s2))
    [NL9]  length (emptyList)             =  0
    [NL10] length (mkList (m))            =  1            when m ≠ errorNatural
    [NL11] length (concat (s1, s2))       =  add (length (s1), length (s2))
end NatLists
```

5. Define alternate abstract syntax for the expression language by specifying a signature with a module and an Σ-algebra for the signature Σ of Expressions that does the following:

   a) Describes only the structure of an expression, so that the abstract syntax tree for "x * (y + z)" is opr (ide(x), opr (ide(y), ide(z))).

   b) Identifies only the first identifier in an expression, so that the abstract syntax tree for "x * (y + z)" is ide(x).

6. Specify modules for the concrete syntax and the abstract syntax of Wren as described in Chapter 1 and show how its term algebra of the abstract syntax module can be interpreted as a Σ-algebra for the signature of the module for the concrete syntax.

12.4 ALGEBRAIC SEMANTICS FOR WREN

We have seen that there are many aspects in specifying the syntax and semantics of a programming language. In Chapter 1 we studied BNF and its variants; in Chapter 2 we built a lexical analyzer and parser for Wren. Context checking was demonstrated using three approaches: attribute grammars (Chapter 3), two-level grammars (Chapter 4), and denotational semantics (Chapter 9). Programming language semantics has been handled in a variety of ways: self-definition (Chapter 6), translational semantics (Chapter 7), structural operational semantics (Chapter 8), denotational semantics (Chapter 9), and axiomatic semantics (Chapter 11). Each technique has its strengths and weaknesses. For example, denotational semantics can perform type checking and program interpretation, but it does not stress lexical analysis and parsing beyond the abstract production rules. Axiomatic semantics does not deal with lexical analysis, parsing, or type checking. Most techniques rely on knowledge of well-known domains, such as truth values with logical operations or numbers with arithmetic operations.

Of the techniques studied so far, algebraic semantics is perhaps the most versatile in its ability to perform all of the functions mentioned above. Modules can be developed to perform lexical analysis, parsing, type checking and language evaluation. Basic domains, such as truth values, natural numbers, and characters, are carefully specified using fundamental concepts, such as zero and the successor function for natural numbers. The initial algebras constructed as quotient algebras represent the meaning of these domains, apart from the renaming of constants and functions. Because of the length of a complete presentation, we elect not to develop the lexical analyzer and parser for Wren using algebraic specifications. See [Bergstra89] for the missing specification techniques. Rather, we concentrate on showing how the methodology can be used to perform type checking and program interpretation. In particular, we develop the following modules:

- WrenTypes specifies the allowed types for Wren programs.
- WrenValues specifies the permissible value domains.
- WrenASTs specifies the output of the parser, the abstract syntax trees.
- WrenTypeChecker returns a Boolean value resulting from context checking.
- WrenEvaluator interprets a Wren program given an input file.
- WrenSystem calls the evaluator if type checking is successful.

For simplicity, we have limited the domain of arithmetic values to natural numbers. To reduce the complexity of the example, declarations allow only a single identifier. Boolean variables can be declared, but we leave their manipulation as an exercise at the end of this section. We also leave the handling of runtime errors, such as division by zero and reading from an empty file, as an exercise. Since nonterminating programs cause technical difficulties in an algebraic specification, we plan to describe only computations (programs and input) that terminate. We want our equations for the Wren evaluator to be sufficiently complete; that is, every program and input list can be reduced to an equivalent term in the sort of lists of natural numbers (output lists). We lose the property of sufficient completeness when we include configurations that produce nonterminating computations.

Types and Values in Wren

The first module, WrenTypes, specifies four constant functions, naturalType, booleanType, programType, and errorType, along with a single Boolean operation to test the equality of two types.

```
module WrenTypes
    imports Booleans
    exports
        sorts WrenType
        operations
            naturalType    : WrenType
            booleanType    : WrenType
            programType    : WrenType
            errorType      : WrenType
            eq? ( _ , _ )  : WrenType, WrenType → Boolean
    end exports

    variables
        t, t1, t2 : WrenType

    equations
    [Wt1]  eq? (t, t) = true                                    when t≠errorType
    [Wt2]  eq? (t1, t2) = eq? (t2,t1)
    [Wt3]  eq? (naturalType, booleanType)   = false
    [Wt4]  eq? (naturalType, programType)   = false
    [Wt5]  eq? (naturalType, errorType)     = false
    [Wt6]  eq? (booleanType, programType)   = false
    [Wt7]  eq? (booleanType, errorType)     = false
    [Wt8]  eq? (programType, errorType)     = false
end WrenTypes
```

The next module, WrenValues, specifies three functions for identifying natural numbers, Boolean values, and an error value. These function symbols perform the same role as the tags in a disjoint union. Two values are equal only if they come from the same domain and if they are equal in that domain.

```
module WrenValues
    imports Booleans, Naturals
    exports
        sorts WrenValue
        operations
            wrenValue ( _ )  :  Natural → WrenValue
            wrenValue ( _ )  :  Boolean → WrenValue
            errorValue       :  WrenValue
            eq? ( _ , _ )    :  WrenValue, WrenValue → Boolean
    end exports
    variables
        x, y        : WrenValue
        n, n1, n2   : Natural
        b, b1, b2   : Boolean
    equations
    [Wv1]   eq? (x, y) = eq? (y,x)
    [Wv2]   eq? (wrenValue(n1), wrenValue(n2))  = eq? (n1,n2)
    [Wv3]   eq? (wrenValue(b1), wrenValue(b2))  = eq? (b1,b2)
    [Wv4]   eq? (wrenValue(n), wrenValue(b))    = false
                                   when m ≠ errorNatural, b ≠ errorBoolean
    [Wv5]   eq? (wrenValue(n), errorValue)      = false    when n ≠ errorNatural
    [Wv6]   eq? (wrenValue(b), errorValue)      = false    when b ≠ errorBoolean
end WrenValues
```

Abstract Syntax for Wren

The abstract syntax tree module specifies the form of a Wren program that has been parsed successfully. As noted previously, we show only the structure of the abstract syntax trees, not how they are constructed.

```
module WrenASTs
    imports Naturals, Strings, WrenTypes
    exports
        sorts  WrenProgram, Block, DecSeq, Declaration,
               CmdSeq, Command,  Expr, Ident
        operations
            astWrenProgram ( _ , _ ) :  Ident, Block           → WrenProgram
            astBlock ( _ , _ )       :  DecSeq, CmdSeq         → Block
            astDecs ( _ , _ )        :  Declaration, DecSeq    → DecSeq
            astEmptyDecs             :  DecSeq
            astDec ( _ , _ )         :  Ident, WrenType        → Declaration
            astCmds ( _ , _ )               : Command, CmdSeq          → CmdSeq
            astOneCmd ( _ )                 : Command                  → CmdSeq
            astRead ( _ )                   : Ident                    → Command
            astWrite ( _ )                  : Expr                     → Command
            astAssign ( _ , _ )             : Ident, Expr              → Command
            astSkip                         : Command
            astWhile ( _ , _ )              : Expr, CmdSeq             → Command
            astIfThen ( _ , _ )             : Expr, CmdSeq             → Command
            astIfElse ( _ , _ , _ )         : Expr, CmdSeq, CmdSeq     → Command
            astAddition ( _ , _ )               : Expr, Expr  → Expr
            astSubtraction ( _ , _ )            : Expr, Expr  → Expr
            astMultiplication ( _ , _ )         : Expr, Expr  → Expr
            astDivision ( _ , _ )               : Expr, Expr  → Expr
            astEqual ( _ , _ )                  : Expr, Expr  → Expr
            astNotEqual ( _ , _ )               : Expr, Expr  → Expr
            astLessThan ( _ , _ )               : Expr, Expr  → Expr
            astLessThanEqual ( _ , _ )          : Expr, Expr  → Expr
            astGreaterThan ( _ , _ )            : Expr, Expr  → Expr
            astGreaterThanEqual ( _ , _ )       : Expr, Expr  → Expr
            astVariable ( _ )                   : Ident       → Expr
            astNaturalConstant ( _ )            : Natural     → Expr
            astIdent ( _ )                      : String      → Ident
    end exports
end WrenASTs
```

If we define a module for the concrete syntax of Wren based on its BNF specification, an algebra modeling WrenASTs will be a homomorphic image of the term algebra over that concrete syntax.

A Type Checker for Wren

The WrenTypeChecker module exports an overloaded function check that returns a Boolean result indicating if the context conditions are satisfied. Calling check with a declaration sequence performs an additional vital function: It builds the symbol table that associates names with types.

```
module WrenTypeChecker
    imports Booleans, WrenTypes, WrenASTs,
            instantiation of Mappings
                bind Entries using String for Domain
                             using WrenType for Range
                             using eq? for equals
```

```
                          using errorString for errorDomain
                          using errorType for errorRange
               rename  using SymbolTable for Mapping
                       using nullSymTab for emptyMap
exports
    operations
        check ( _ )       : WrenProgram             → Boolean
        check ( _ , _ )   : Block, SymbolTable      → Boolean
        check ( _ , _ )   : DecSeq, SymbolTable     → Boolean, SymbolTable
        check ( _ , _ )   : Declaration, SymbolTable → Boolean, SymbolTable
        check ( _ , _ )   : CmdSeq, SymbolTable     → Boolean
        check ( _ , _ )   : Command, SymbolTable    → Boolean
end exports

operations
    typeExpr : Expr, SymbolTable → WrenType

variables
    block                       : Block
    decs                        : DecSeq
    dec                         : Declaration
    cmds, $cmds_1$, $cmds_2$    : CmdSeq
    cmd                         : Command
    expr, $expr_1$, $expr_2$    : Expr
    type                        : WrenType
    symtab, $symtab_1$          : SymbolTable
    m                           : Natural
    name                        : String
    b, $b_1$, $b_2$             : Boolean

equations
[Tc1]   check (astWrenProgram (astIdent (name), block))
            = check (block, update(nullSymTab, name, programType))

[Tc2]   check (astBlock (decs, cmds), symtab)
            = and ($b_1$,$b_2$)
                      when  <$b_1$,$symtab_1$> = check (decs, symtab),
                             $b_2$ = check (cmds, $symtab_1$),

[Tc3]   check (astDecs (dec, decs), symtab)
            = <and ($b_1$,$b_2$), $symtab_2$>
                      when  <$b_1$,$symtab_1$> = check (dec, symtab),
                            <$b_2$,$symtab_2$> = check (decs, $symtab_1$)
```

```
[Tc4]  check (astEmptyDecs, symtab)
           = <true, symtab>
[Tc5]  check (astDec (astIdent (name), type), symtab)
           = if ( apply (symtab, name) = errorType,
                  <true, update(symtab, name, type)>,
                  <false, symtab>)
[Tc6]  check (astCmds (cmd, cmds), symtab)
           = and (check (cmd, symtab), check (cmds, symtab))
[Tc7]  check (astOneCmd (cmd), symtab)
           = check (cmd, symtab)
[Tc8]  check (astRead (astIdent (name)), symtab)
           = eq?(apply (symtab, name), naturalType)
[Tc9]  check (astWrite (expr, symtab)
           = eq? (typeExpr (expr, symtab), naturalType)
[Tc10] check (astAssign (astIdent (name), expr), symtab)
           = eq? (apply(symtab, name), typeExpr (expr, symtab))
[Tc11] check (astSkip, symtab)
           = true
[Tc12] check (astWhile (expr, cmds), symtab)
           = if ( eq? (typeExpr (expr, symtab), booleanType),
                  check (cmds, symtab),
                  false)
[Tc13] check (astIfThen (expr, cmds), symtab)
           = if ( eq? (typeExpr (expr, symtab), booleanType),
                  check (cmds, symtab),
                  false)
[Tc14] check (astIfElse (expr, cmds1, cmds2), symtab)
           = if ( eq? (typeExpr (expr, symtab), booleanType),
                  and (check (cmds1, symtab), check (cmds2, symtab)),
                  false)
[Tc15] typeExpr (astAddition (expr1, expr2), symtab)
           = if ( and (eq? (typeExpr (expr1, symtab), naturalType),
                               eq? (typeExpr (expr2, symtab), naturalType)),
                  naturalType,
                  errorType)
```

```
[Tc16] typeExpr (astSubtraction (expr₁, expr₂), symtab)
           = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  naturalType,
                  errorType)
[Tc17] typeExpr (astMultiplication (expr₁, expr₂), symtab)
           = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  naturalType,
                  errorType)
[Tc18] typeExpr (astDivision (expr₁, expr₂), symtab)
           = if ( and(eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  naturalType,
                  errorType)
[Tc19] typeExpr (astEqual (expr₁, expr₂), symtab)
           = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  booleanType,
                  errorType)
[Tc20] typeExpr (astNotEqual (expr₁,expr₂), symtab)
           = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  booleanType,
                  errorType)
[Tc21] typeExpr (astLessThan (expr₁, expr₂), symtab)
           = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  booleanType,
                  errorType)
[Tc22] typeExpr (astLessThanEqual (expr₁, expr₂), symtab)
           = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  booleanType,
                  errorType)
[Tc23] typeExpr (astGreaterThan (expr₁,expr₂),symtab)
           = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                              eq? (typeExpr (expr₂, symtab), naturalType)),
                  booleanType,
                  errorType)
```

```
    [Tc24]  typeExpr (astGreaterThanEqual (expr₁, expr₂), symtab)
                = if ( and (eq? (typeExpr (expr₁, symtab), naturalType),
                                    eq? (typeExpr (expr₂, symtab), naturalType)),
                      booleanType,
                      errorType)
    [Tc25]  typeExpr (astNaturalConstant (m), symtab)
                = naturalType
    [Tc26]  typeExpr (astVariable (astIdent(name)), symtab)
                = apply (symtab, name)
end WrenTypeChecker
```

Most of the type-checking equations are self-evident; we point out only general features here. Equations [Tc1], [Tc3], and [Tc5] build the symbol table from the declarations while ensuring that no identifier is declared twice. [Tc1] adds the program name with programType to the table. Most of the equations for commands pass the symbol table information along for checking at lower levels. The following equations perform the actual type checking:

[Tc5]	No identifier is declared more than once.
[Tc8]	The variable in a **read** command has naturalType.
[Tc9]	The expression in a **write** command has naturalType.
[Tc10]	The assignment target variable and expression have the same type.
[Tc12-14]	The expressions in **while** and **if** commands have booleanType.
[Tc15-18]	Arithmetic operations involve expressions of naturalType.
[Tc19-24]	Comparisons involve expressions of naturalType.

An Interpreter for Wren

The WrenEvaluator module is used to specify semantic functions that give meaning to the constructs of Wren. The top-level function meaning takes a Wren program and an input file and returns the output file resulting from executing the program. We assume that the output file is initially empty. The declaration sequence builds a store that associates each declared variable with an initial value, zero for naturalType and false for booleanType. Commands use the current store, input file, and output file to compute a new store, a new input file, and a new output file. Evaluating an expression produces a WrenValue.

```
module WrenEvaluator
    imports Booleans, Naturals, Strings, Files, WrenValues, WrenASTs,
              instantiation of Mappings
                bind Entries using String for Domain
                             using WrenValue for Range
                             using eq? for equals
                             using errorString for errorDomain
                             using errorValue for errorRange
                rename  using Store for Mapping
                        using emptySto for emptyMap
                        using updateSto for update
                        using applySto for apply
    exports
        operations
            meaning ( _ , _ )           : WrenProgram, File           → File
            perform ( _ , _ )           : Block, File                 → File
            elaborate ( _ , _ )         : DecSeq, Store               → Store
            elaborate ( _ , _ )         : Declaration, Store          → Store
            execute ( _ , _ , _ , _ )   : CmdSeq, Store, File, File   → Store, File, File
            execute ( _ , _ , _ , _ )   : Command, Store, File, File  → Store, File, File
            evaluate ( _ , _ )          : Expr, Store                 → WrenValue
    end exports

    variables
        input, input1, input2         : File
        output, output1, output2      : File
        block                         : Block
        decs                          : DecSeq
        cmds, cmds1, cmds2            : CmdSeq
        cmd                           : Command
        expr, expr1, expr2            : Expr
        sto, sto1, sto2               : Store
        value                         : WrenValue
        m,n                           : Natural
        name                          : String
        b                             : Boolean

    equations
    [Ev1]   meaning (astWrenProgram (astIdent (name), block), input)
                = perform (block, input)

    [Ev2]   perform (astBlock (decs,cmds), input)
                = execute (cmds, elaborate (decs, emptySto), input, emptyFile)
```

```
[Ev3]  elaborate (astDecs (dec, decs), sto)
           = elaborate (decs,elaborate(dec, sto))
[Ev4]  elaborate (astEmptyDecs, sto)
           = sto
[Ev5]  elaborate (astDec (astIdent (name), naturalType), sto)
           = updateSto(sto, name, wrenValue(0))
[Ev6]  elaborate (astDec (astIdent (name), booleanType), sto)
           = updateSto(sto, name, wrenValue(false))
[Ev7]  execute (astCmds (cmd, cmds), sto_1, input_1, output_1)
           = execute (cmds, sto_2, input_2, output_2)
           when <sto_2, input_2, output_2> = execute (cmd, sto_1, input_1, output_1)
[Ev8]  execute (astOneCmd (cmd), sto, input, output)
           = execute (cmd, sto, input, output)
[Ev9]  execute (astSkip, sto, input, output)
           = <sto, input, output>
[Ev10] execute (astRead(astIdent (name)), sto, input, output)
           = if (empty? (input),
                 error case left as an exercise
                 <updateSto(sto, name, first), rest, output>)
                                  when input = cons(first,rest)
[Ev11] execute (astWrite (expr), sto, input, output)
           = <sto, input, concat (output, mkFile (evaluate (expr, sto)))>
[Ev12] execute (astAssign (astIdent (name), expr), sto, input, output)
           = <updateSto(sto, name, evaluate (expr, sto)), input, output>
[Ev13] execute (astWhile (expr, cmds), sto_1, input_1, output_1)
           = if (eq? (evaluate (expr, sto_1), wrenValue(true))
                 execute (astWhile(expr, cmds), sto_2, input_2, output_2)
                           when <sto_2, input_2, output_2> =
                                       execute (cmds, sto_1, input_1, output_1),
                 <sto_1, input_1, output_1>)
[Ev14] execute (astIfThen (expr, cmds), sto, input, output)
           = if (eq? (evaluate (expr, sto), wrenValue(true))
                 execute (cmds, sto, input, output),
                 <sto, input, output>)
[Ev15] execute (astIfElse (expr, cmds_1, cmds_2), sto, input, output)
           = if (eq? (evaluate (expr, sto), wrenValue(true))
                 execute (cmds_1, sto, input, output)
                 execute (cmds_2, sto, input, output))
```

```
[Ev16]  evaluate (astAddition (expr₁, expr₂), sto)
            = wrenValue(add (m,n))
            when  wrenValue(m) = evaluate (expr₁, sto),
                  wrenValue(n) = evaluate (expr₂, sto)
[Ev17]  evaluate (astSubtraction (expr₁, expr₂), sto)
            = wrenValue(sub (m,n))
            when  wrenValue(m) = evaluate (expr₁, sto),
                   wrenValue(n) = evaluate (expr₂, sto)
[Ev18]  evaluate (astMultiplication (expr₁, expr₂), sto)
            = wrenValue(mul (m,n))
            when  wrenValue(m) = evaluate (expr₁, sto),
                   wrenValue(n) = evaluate (expr₂, sto)
[Ev19]  evaluate (astDivision (expr₁, expr₂), sto)
            = wrenValue(div (m,n))
            when  wrenValue(m) = evaluate (expr₁, sto),
                   wrenValue(n) = evaluate (expr₂, sto)
[Ev20]  evaluate (astEqual (expr₁, expr₂), sto)
            = wrenValue(eq? (m,n))
            when  wrenValue(m) = evaluate (expr₁, sto),
                   wrenValue(n) = evaluate (expr₂, sto)
[Ev21]  evaluate (astNotEqual (expr₁, expr₂), sto)
            = wrenValue(not (eq? (m,n)))
            when  wrenValue(m) = evaluate (expr₁, sto),
                   wrenValue(n) = evaluate (expr₂, sto)
[Ev22]   evaluate (astLessThan (expr₁, expr₂), sto)
            = wrenValue(less? (m,n))
            when  wrenValue(m) = evaluate (expr₁, sto),
                  wrenValue(n) = evaluate (expr₂, sto)
[Ev23]   evaluate (astLessThanEqual (expr₁, expr₂), sto)
            = wrenValue(not(greater? (m,n)))
            when  wrenValue(m) = evaluate (expr₁, sto),
                  wrenValue(n) = evaluate (expr₂, sto)
[Ev24]   evaluate (astGreaterThan (expr₁, expr₂), sto)
            = wrenValue(greater? (m,n))
            when  wrenValue(m) = evaluate (expr₁, sto),
                  wrenValue(n) = evaluate (expr₂, sto)
```

```
[Ev25]  evaluate (astGreaterThanEqual (expr₁, expr₂), sto)
             = wrenValue(not(less? (m,n)))
             when  wrenValue(m) = evaluate (expr₁, sto),
                   wrenValue(n) = evaluate (expr₂, st)

[Ev26]  evaluate (astNaturalConstant (m), sto)
             = wrenValue(m)

[Ev27]  evaluate (astVariable (astIdent (name)), sto)
             = applySto (sto, name)
end WrenEvaluator
```

Each equation should be self-explanatory. Observe that [Ev10] is incomplete, as we have only indicated that error handling is needed for reading from an empty file. Also [Ev17] might cause an error when a larger number is subtracted from a smaller number, or [Ev19] when any number is divided by zero. We have elected not to show this error handling since it requires modifications to almost all equations to propagate the error to the top level, and this introduces unwanted complexity. Two exercises deal with alternative error-handling techniques.

A Wren System

Our final module, WrenSystem, invokes the type checker and, if it succeeds, calls the evaluator. If type checking fails, the empty file is returned. Remember that we have assumed the program interpretation completes successfully to avoid technical issues relating to sufficient completeness.

```
module WrenSystem
    imports WrenTypeChecker, WrenEvaluator
    exports
        operations
            runWren : WrenProgram, File → File
    end exports

    variables
        input    : File
        program  : WrenProgram

    equations
    [Ws1]   runWren (program, input)  = if (check (program),
                                             meaning (program, input),
                                             emptyFile)
            -- return an empty file if context violation, otherwise run program
end WrenSystem
```

This completes the development of an algebraic specification for Wren. In the next section, we implement part of this specification in Prolog.

Exercises

1. What changes, if any, would be needed in the modules presented in this section if an Integers module were used in place of a Naturals module?

2. Complete the syntactic and semantic functions and equations for Boolean expressions. The comparisons given in this section will be only one possible alternative for a Boolean expression.

3. One technique of error handling is to assign default values such as zero when an item is read from an empty file. For division by zero, consider introducing a constant representing a maximum allowed natural number. Assuming WordSize is imported from a module called ComputerSystem, how can such a value be defined? Indicate by revising the equations how all arithmetic operations have to guard against exceeding such a value.

4. Halting evaluation due to a fatal runtime error, such as reading from an empty file or division by zero, is difficult to specify. Briefly indicate how this problem can be handled by returning a Boolean value (in addition to other values) to indicate whether each operation is successful.

12.5 LABORATORY: IMPLEMENTING ALGEBRAIC SEMANTICS

As with other semantic definitions, algebraic specifications can be translated directly into Prolog. In the development presented in this section, we assume that the lexical analyzer and parser given in Chapter 2 provide input to the interpreter. The user is asked to specify a file containing a Wren program and an input file (a list of natural numbers). Interpreting the program produces the output file. Numerals in the input file are translated into natural number notation for processing, and when a **write** statement is encountered, values in natural number notation are translated to base-ten numerals.

We show the implementation of three modules: Booleans, Naturals, and WrenEvaluator. We have not translated identifier names into the Strings notation based on a Characters module; these modules are left as an exercise at the end of this section. Implementation of the modules for Files and Mappings is also left as an exercise. Observe that we have no mechanism for implementing generic modules, such as Lists, in Prolog, so we simply implement the instantiated modules directly. Finally, we have not implemented the WrenTypeChecker; that project has also been left as an exercise.

Before examining the implementations of the modules, we inspect the expected behavior of the system, including the output of the parser to remind us of the format produced by the language-processing system. This program converts a list of binary digits into the corresponding decimal number using any integer greater than 1 to terminate the list.

```
>>> Interpreting Wren via Algebraic Semantics <<<
Enter name of source file: frombinary.wren
    program frombinary is
      var sum,n : integer;
    begin
      sum := 0; read n;
      while n<2 do
        sum := 2*sum+n; read n
       end while;
      write sum
    end
Scan successful
Parse successful
prog([dec(integer,[sum,n])],
     [assign(sum,num(0)),read(n),
      while(exp(less,ide(n),num(2)),
       [assign(sum,exp(plus,exp(times,num(2),ide(sum)),ide(n))),
           read(n)]),
      write(ide(sum))])
Enter an input list followed by a period: [1,0,1,0,1,1,2].
Output = [43]
yes
```

Module Booleans

The implementation of the module Booleans includes the constants true and false and the functions not, and, or, xor, and beq (note the name change to avoid confusion with equality in the Naturals module).

```
boolean(true).
boolean(false).

bnot(true, false).
bnot(false, true).
```

```
and(true, P, P).
and(false, true, false).
and(false, false, false).

or(false,P,P).
or(true,P,true) :- boolean(P).

xor(P, Q, R) :- or(P,Q,PorQ), and(P,Q,PandQ),
                    bnot(PandQ,NotPandQ), and(PorQ,NotPandQ, R).

beq(P, Q, R) :- xor(P,Q,PxorQ), bnot(PxorQ,R).
```

We have followed the specifications given in the module Booleans closely except for the direct definition of or. We misspell not as bnot to avoid conflict with the predefined predicate for logical negation that may exist in some Prolog implementations.

Module Naturals

The implementation of Naturals follows directly from the algebraic specification. The predicate natural succeeds with arguments of the form

zero, succ(zero), succ(succ(zero)), succ(succ(succ(zero))), and so on.

Calling this predicate with a variable, such as natural(M), generates the natural numbers in this form if repeated solutions are requested by entering a semicolon after each successful answer to the query.

```
natural(zero).
natural(succ(M)) :- natural(M).
```

The arithmetic functions follow the algebraic specification. Rather than return an error value for subtraction of a larger number from a smaller number or for division by zero, we print an appropriate error message and abort the program execution. The comparison operations follow directly from their definitions. Observe how the conditions in the specifications are handled in the Prolog clauses. We give a definition of the exponentiation operation now for completeness.

```
add(M, zero, M) :- natural(M).
add(M, succ(N), succ(R)) :- add(M,N,R).

sub(zero, succ(N), R) :- write('Error: Result of subtraction is negative'), nl, abort.
sub(M, zero, M) :- natural(M).
sub(succ(M), succ(N), R) :- sub(M,N,R).
```

```
mul(M, zero, zero) :- natural(M).
mul(M, succ(zero), M) :- natural(M).
mul(M, succ(succ(N)), R)  :- mul(M,succ(N),R1), add(M,R1,R).

div(M, zero, R) :- write('Error: Division by zero'), nl, abort.
div(M, succ(N), zero) :- less(M,succ(N),true).
div(M,succ(N),succ(Quotient)) :- less(M,succ(N),false),
                                  sub(M,succ(N),Dividend),
                                  div(Dividend,succ(N),Quotient).

exp(succ(M), zero, succ(zero)) :- natural(M).
exp(M, succ(zero), M) :- natural(M).
exp(M, succ(N), R) :- exp(M,N,MexpN), mul(M, MexpN, R).

eq(zero,zero,true).
eq(zero,succ(N),false) :- natural(N).
eq(succ(M),zero,false) :- natural(M).
eq(succ(M),succ(N),BoolValue) :- eq(M,N,BoolValue).

less(zero,succ(N),true) :- natural(N).
less(M,zero,false) :- natural(M).
less(succ(M),succ(N),BoolValue) :- less(M,N,BoolValue).

greater(M,N,BoolValue) :- less(N,M,BoolValue).

lesseq(M,N,BoolValue) :- less(M,N,B1), eq(M,N,B2), or(B1,B2,BoolValue).

greatereq(M,N,BoolValue) :- greater(M,N,B1), eq(M,N,B2), or(B1,B2,BoolValue).
```

We add two operations not specified in the Naturals module that convert base-ten numerals to natural numbers as defined in the module Naturals using successor notation and vice versa. Specifically, toNat converts a numeral to natural notation and toNum converts a natural number to a base-ten numeral. For example, toNat(4,Num) returns Num = succ (succ (succ (succ (zero)))).

```
toNat(0,zero).
toNat(Num, succ(M)) :- Num>0, NumMinus1 is Num-1, toNat(NumMinus1, M).

toNum(zero,0).
toNum(succ(M),Num) :- toNum(M,Num1), Num is Num1+1.
```

Declarations

The clauses for elaborate are used to build a store with numeric variables initialized to zero and Boolean variables initialized to false.

```
elaborate([Dec|Decs],StoIn,StoOut) :-                                   % Ev3
                                   elaborate(Dec,StoIn,Sto),
                                   elaborate(Decs,Sto,StoOut).

elaborate([ ],Sto,Sto).                                                 % Ev4

elaborate(dec(integer,[Var]),StoIn,StoOut) :-                           % Ev5
                                   updateSto(StoIn,Var,zero,StoOut).

elaborate(dec(boolean,[Var]),StoIn,StoOut) :-                           % Ev6
                                   updateSto(StoIn,Var,false,StoOut).
```

Commands

For a sequence of commands, the commands following the first command are evaluated with the store produced by the first command. The Prolog code is simpler if we allow an empty command sequence as the base case.

```
execute([Cmd|Cmds],StoIn,InputIn,OutputIn,                              %Ev7
                                        StoOut,InputOut,OutputOut) :-
          execute(Cmd,StoIn,InputIn,OutputIn,Sto,Input,Output),
          execute(Cmds,Sto,Input,Output,StoOut,InputOut,OutputOut).

execute([ ],Sto,Input,Output,Sto,Input,Output).                         % Ev8
```

The **read** command removes the first item from the input file, converts it to natural number notation, and places the result in the store. The **write** command evaluates the expression, converts the resulting value from natural number notation to a numeric value, and appends the result to the end of the output file.

```
execute(read(Var),StoIn,emptyFile,Output,StoOut,_,Output) :-           % Ev10
            write('Fatal Error: Reading an empty file'), nl, abort.

execute(read(Var),StoIn,[FirstIn|RestIn],Output,StoOut,RestIn,Output) :- % Ev10
            toNat(FirstIn,Value),
            updateSto(StoIn,Var,Value,StoOut).

execute(write(Expr),Sto,Input,OutputIn,Sto,Input,OutputOut) :-         % Ev11
            evaluate(Expr,StoIn,ExprValue),
            toNum(ExprValue,Value),
            mkFile(Value,ValueOut),
            concat(OutputIn,ValueOut,OutputOut).
```

Assignment evaluates the expression using the current store and then updates that store to reflect the new binding. The **skip** command makes no changes to the store or to the files.

```
    execute(assign(Var,Expr),StoIn,Input,Output,StoOut,Input,Output) :-     % Ev12
                evaluate(Expr,StoIn,Value).
                updateSto(StoIn,Var,Value,StoOut).

    execute(skip,Sto,Input,Output,Sto,Input,Output).                          % Ev9
```

The two forms of **if** commands test the Boolean expressions and then let a predicate select carry out the appropriate actions. Observe how the one-alternative **if** command passes an empty command sequence to select. If the comparison in the **while** command is false, the store and files are returned unchanged. If the comparison is true, the **while** command is reevaluated with the store and files resulting from the execution of the **while** loop body. These commands are implemented with auxiliary predicates, select and iterate, to minimize the amount of backtracking the system must do.

```
  execute(if(Expr,Cmds),StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut) :-
    evaluate(Expr,StoIn,BoolVal),                                             % Ev14
    select(BoolVal,Cmds,[ ],StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut).

  execute(if(Expr,Cmds1,Cmds2),StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut) :-
    evaluate(Expr,StoIn,BoolVal),                                             % Ev15
    select(BoolVal,Cmds1,Cmds2,StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut).

  select(true,Cmds1,Cmds2,StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut) :-
    execute(Cmds1,StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut).

  select(false,Cmds1,Cmds2,StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut) :-
    execute(Cmds2,StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut).

  execute(while(Expr,Cmds),StoIn,InputIn,OutputIn, StoOut,InputOut,OutputOut) :-
    evaluate(Expr,StoIn,BoolVal),                                             % Ev13
    iterate(BoolVal,Expr,Cmds,StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut).

  iterate(true,Expr,Cmds,StoIn,InputIn,OutputIn,StoOut,InputOut,OutputOut) :-
    execute(Cmds,StoIn,InputIn,OutputIn,Sto,Input,Output),
    execute(while(Expr,Cmds),Sto,Input,Output,StoOut,InputOut,OutputOut).

  iterate(false,Expr,Cmds,Sto,Input,Output,Sto,Input,Output).
```

Expressions

The evaluation of arithmetic expressions is straightforward. Addition is shown below; the other three operations are left as exercises. Evaluating a variable involves looking up the value in the store. A numeric constant is converted to natural number notation and returned.

```
evaluate(exp(plus,Expr1,Expr2),Sto,Result) :-                    % Ev16
                        evaluate(Expr1,Sto,Val1),
                        evaluate(Expr2,Sto,Val2),
                        add(Val1,Val2,Result).

evaluate(num(Constant),Sto,Value) :- toNat(Constant,Value).      %Ev26

evaluate(ide(Var),Sto,Value) :- applySto(Sto,Var,Value).          % Ev27
```

Evaluation of comparisons is similar to arithmetic expressions; the equal comparison is given below, and the five others are left as an exercise.

```
evaluate(exp(equal,Expr1,Expr2),Sto,Bool) :-                     % Ev20
                        evaluate(Expr1,Sto,Val1),
                        evaluate(Expr2,Sto,Val2),
                        eq(Val1,Val2,Bool).
```

The Prolog implementation of the algebraic specification of Wren is similar to the denotational interpreter with respect to command and expression evaluation. Perhaps the biggest difference is in not relying on Prolog native arithmetic to perform comparisons and numeric operations. Instead, the Naturals module performs these operations based solely on a number system derived from applying a successor operation to an initial value zero.

More elaborate approaches are possible in which the original specification module is read and interpreted. These tasks are beyond the scope of this book; interested readers can consult the further readings at the end of this chapter.

Exercises

1. Complete the interpreter as presented by adding the modules Files and Mappings and by completing the remaining arithmetic operations and comparison operations.

2. As an extension of exercise 2 in section 12.4, implement the syntactic and semantic functions and equations for Boolean expressions in Prolog.

3. Add modules for Characters and Strings. Translate identifiers from the parser, such as ide(name), into Strings, such as cons(char-n, cons(char-a, cons(char-m, cons(char-e, nullString)))).

4. Implement the modules WrenTypeChecker and WrenSystem. If a context violation is encountered, print an appropriate error message, indicating where the error occurred. Process the remainder of the program for other context violations, but do not evaluate the program.

5. Change the Naturals module to be an Integers module. Be sure to change other parts of the program, such as removing the error on subtraction, accordingly.

12.6 FURTHER READING

Ehrig and Mahr [Ehrig85] present the best overall discussion of algebraic specifications and the algebras that model them with a clear presentation of the theory. This subject matter developed from the work done on abstract data types by the ADJ group in the 1970s [Goguen78]. Watt's book on formal semantics [Watt91] also serves as a good introduction to algebraic specifications supported by many examples. The short paper by Burstall and Goguen [Burstall82] provides a concise but well-motivated discussion of specifications. For a more advanced treatment of the subject, see [Wirsing90].

The algebraic specification of data types has been developed primarily by John Guttag and the ADJ group. The best presentations of abstract data types can be found in [Guttag78a], [Guttag78b], [Guttag78c], and [Guttag80]. [Goguen77] and [Broy87] both discuss the use of algebraic specifications to model abstract syntax. For more on abstract syntax, see [Noonan85] and [Pagan83].

Using algebraic methods to specify the semantics of a programming language is covered in considerable detail in [Bergstra89]. Our specification of Wren is largely based on the ideas in his book. Another presentation of algebraic semantics can be found in [Broy87].