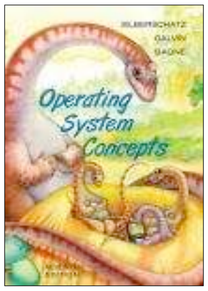
Syllabus for the Ph.D. Prelim Exam

Revised August 15, 2009

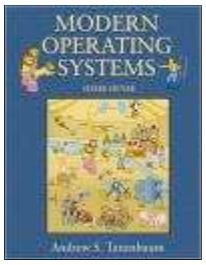
Operating Systems

Reference Texts:

1. Silberschatz, Galvin, and Gagne, Operating System Concepts, 7th ed., John Wiley & Sons, 2004. Chapter 1 – 14



2. Tanenbaum, Woodhull, Modern Operating Systems, 2nd ed., Prentice-Hall. Chapters 1 - 6



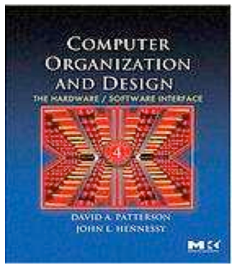
Topics:

1. Basics
   a. Common hardware support for OS
   b. Interrupt-driven nature of OS.
   c. Hardware supported protection
   d. I/O buffering and SPOOL
2. OS architecture
3. Process management and scheduling
4. Concurrency control
5. Deadlock handling
6. Memory management
7. I/O scheduling
8. File system basics.

Computer Architecture

Reference Text:

1. David A. Patterson & John L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 4th Ed., Morgan Kaufmann, 2009.



Topics:

1. Introduction, Performance (Chapter 1)
2. Instructions (Chapter 2)
3. Computer Arithmetic (division operations excluded) (Chapter 3)
4. Processor: Data-path (Chapter 4)
5. Processor: Control (Chapter 4 & Appendix D)
6. Pipelining Techniques (Chapter 4)
7. Memory (Chapter 5)

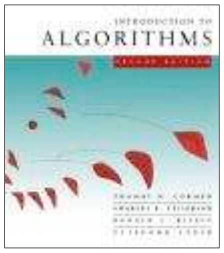
8. Parallelism and Multiprocessors (definitions & intro level materials only) (Chapter 7)

Materials related to assembly language are based on MIPS instruction set as in the text. Knowledge of other specific machines such as AMD Opteron X4, ARM, x86, and Intel Nehalem, are not required.

Algorithms and Data Structures

Reference Text:

1. T. Cormen, R. Rivest , C. Leiserson, and C. Stein, Introduction to Algorithms, 2nd ed., MIT Press and McGraw-Hill, 2001



Topics:

1. Mathematical Foundations
2. Sorting and Order Statistics
3. Data Structures
4. Dynamic Programming, Greedy Algorithms, Amortized Analysis
5. Graph Algorithms
6. NP-Completeness

There are standard DSs that should be known to any computer engineer (stacks, queues, trees, tables, arrays, graphs, etc. - several varieties of several of these - priority, binary, ordered, balanced, hashed, pseudo, multi, etc.), they are covered in nearly every DS book. The distinctions in books tend to be in their treatment of Abstract Data Types - a mathematical notion. It is important to understand what an abstract data type (ADT) is. Some books do nothing but give lip service to the specification of an ADT (which should be done formally, but can be done informally independent of any programming language or constructs).

It is important to know the distinctions between specification, representation, implementation, and application of data structures, and to be able to design appropriate data structures suitable to a problem - understanding tradeoffs in the choices made.

To help supplement the usual textbook specifications, which may or may not be clear, the following pages provide some notes on ADTs using the Larch Shared Language as specification medium.

Formal Specifications and Advanced Data Structures

There is more to an abstract data structure than a set of elements. A data structure is defined by the elements of its domain together with primitives that describe how these elements are related to each other and to elements of other data structures. We can use the following definition to express this more formally.

DEFINITION: A *data structure* is a set of domains $\mathcal{D}$, a designated domain $d \in \mathcal{D}$, a set of functions $\mathcal{F}$, and a set of axioms $\mathcal{A}$. The 4-tuple $(\mathcal{D}, d, \mathcal{F}, \mathcal{A})$ denotes the data structure d and is usually abbreviated by writing d. ∎

It is important to distinguish between data structures, which are abstract entities, and the storage structures that are provided in a programming language, even though they may use the same name. Even a structure as simple as an array is not necessarily what it seems from programming experience. The important quality of an array is that it is a mapping from indices to values. To the user this simply means that given an index one can store or retrieve an associated value. An array is usually represented by a sequential chunk of memory and the mapping is implemented by arithmetic on machine addresses. However, one may in fact choose to represent an array differently, perhaps allowing dynamic extension or shrinking of the space required to store the elements. The choice of representation and algorithms used to implement an array are irrelevant to a user, who simply wants to make use of its abstract nature as an association of indices with values.

A data structure is *specified* to describe *what it is* (its elements and the relationships among its elements and those of other data structures). An *abstract representation* of a data structure provides a way to denote elements of the structure by combinations of other structures. A *concrete representation* denotes elements of an ADT by combinations of storage structures in a specific programming language. (While we define both concepts below, in practice we often move from a specification directly to a concrete representation.) An *implementation* is a set of declarations and programs defining the elements and primitive operations that provide the capabilities of the functions specified for the the data structure (possibly as well as other utilities, e.g., I/O), using a concrete representation.

DEFINITION: An (abstract/concrete) *representation* of a data structure d is a mapping from d to a set of other (data structures/storage structures) $\mathcal{E}$, such that:

1. This mapping describes how every element of d can be represented by some element(s) of $\mathcal{E}$.
2. Given a representation, $\mathbf{r}$ in $\mathcal{E}$, of an element of d, we must be able to unambiguously determine the element in d represented by $\mathbf{r}$. ∎

There may be more than one way to represent a given element of a data domain. For example, if our domain is the set of finite sets of integers, we might choose to represent a given set by an unordered list containing all of the elements in that set. There may be several different lists that all represent the same set. For example,

(1, 2, 3)
(2, 1, 3)
(3, 2, 2, 1, 3)

all represent the set containing the first three positive integers, which might be written: {1, 2, 3}. Lists may appropriately contain several copies of the same element, and the order of elements in a list may be important. However, sets are unordered collections and items are either elements of the set or not, it makes no sense to say something is an element two or more times.

The second requirement of the definition ensures that given a list of integers, there can be only one set of integers that it represents. That is, even though there may be several different representations of the same abstract element of the data structure, a given representation can only be interpreted as one abstract element.

DEFINITION: An *implementation* of a data structure d is a collection of subprograms that allow us to operate on the concrete representation in a way that is consistent with the properties of the data structure. If every primitive of d is provided using the operations of $\mathcal{E}$, such that the axioms of d are satisfied, then d is said to be implemented in terms of the structures in $\mathcal{E}$. ■

There are many different specification languages and techniques. A specification is intended to accurately define the *one data structure* of interest. It is abstract in that it describes the values and relationships without concern for how they might be represented and implemented.

There can be many possible representations for a given data structure. A representation tells us how a given element of the data structure looks so we can recognize it and work with it.

Given a specific representation, there can be many possible implementations using different algorithms to accomplish the same task. An implementation tells us how the primitive operations work with the chosen representation to reflect the capabilities of the specified data structure.

The diagram in Figure 1 shows the relationships among specifications, representations, and implementations of an abstract data structure.

The clear division of the tasks of *specification, representation*, and *implementation* of data structures helps to clarify distinct issues that are raised at each step.

1 Specification

To specify an abstract data structure (or *abstract data type*, ADT) we need a convention for defining the domain of elements, primitive functions, and the axioms defining constraints that must be satisfied. We will use the Larch Shared Language (LSL), defined in [?], to describe high-level, programming-language independent specifications. Larch is, in fact, a family of specification languages and tools that support a two-tiered approach to specification of software and hardware. The Larch Shared Language is used to write a programming-

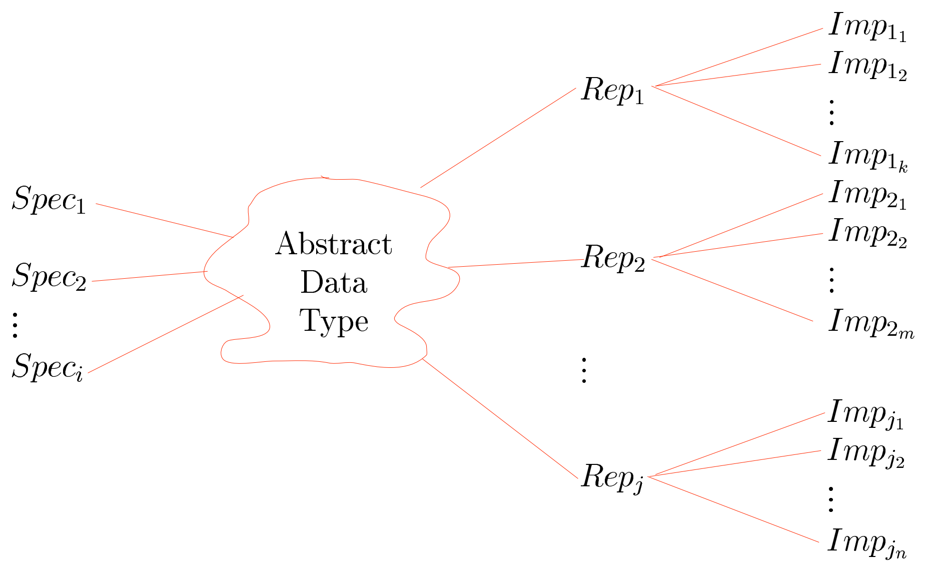


Figure 1: ADT with Specifications, Representations, and Implementations

language independent specification describing mathematical abstractions. The second tier of specification is supported by Larch Interface Languages (LILs) that have been designed for several programming languages (Ada, C, Modula, CLU, Smalltalk, ML, ...), allowing one to specify resource allocation, state changes, and exceptions in a way that depends heavily on the specifics of a particular implementation language. Larch tools include several syntax checkers and the Larch Prover for analyzing the semantics of specifications.

LSL provides a specification unit called a *trait* that can be used to specify (among other options) abstract data structures. For example, we can specify the data structure *NatNum* using a trait of LSL. Using the definition above, the 4-tuple consists of $d = NatNum$, $\mathcal{D} = \{NatNum, Boolean\}$, $\mathcal{F} = \{Zero, Succ, Pred, IsZero, Eq, Add\}$, and $\mathcal{A}$ is the set of axioms following the **asserts** below.

NatNum : **trait**
    introduces
        elements basis $Zero : \rightarrow NatNum$
                    constructor $Succ : NatNum \rightarrow NatNum$
        selectors $Pred : NatNum \rightarrow NatNum$
        testers $IsZero : NatNum \rightarrow Boolean$
                $Eq : NatNum, NatNum \rightarrow Boolean$
        utilities $Add : NatNum, NatNum \rightarrow NatNum$
    asserts
    $NatNum$ **generated by** $Zero, Succ$
    for all $x, y : \; NatNum$
        $Pred(Succ(x)) == x$
        $IsZero(Zero) == true$
        $IsZero(Succ(x)) == false$

$Eq(x, Zero) == IsZero(x)$
$Eq(Zero, Succ(x)) ==$*false*
$Eq(Succ(x), Succ(y)) == Eq(x, y)$
$Add(Zero, y) == y$
$Add(Succ(x), y) == Succ(Add(x, y))$

exempting $Pred(Zero)$

end $NatNum$

The **introduces** section specifies the set of functions $\mathcal{F}$ and provides the types of their arguments and values. The axioms following the **asserts** specify constraints or conditions that must be satisfied by any correct implementation of the data structure. These specify that certain relationships must hold, and do not depend on any particular representation of the data structures involved. Note that there are several ways of specifying the same thing. For example, instead of the second axiom listed for Eq above, we could have written:

$$Eq(Zero, x) == IsZero(x)$$

While it is not important exactly how we specify each constraint, it *is* important that we cover all the cases we wish to constrain. In the case of $NatNum$ we know we have covered every possibility if we cover all combinations of each $NatNum$ argument being either $Zero$ or $Succ(x)$, where x is a $NatNum$.

Note that we did not completely specify the selector $Pred$, since we did not say what happens if $Pred$ is applied to $Zero$. There is no natural number that is the predecessor of $Zero$, but the trait has simply left out any equation defining such a value. It is left up to the implementor whether this should signal an error or return some other value (such as 0). The declaration that we are **exempting** specific applications of an operator explicitly points out the fact that the operator is not completely defined on all elements of the domain by the equations in the trait.

The set of functions $\mathcal{F}$ specifies the elements of the data domain of the structure being defined as well as the primitive operations and relationships involving those elements. There is no fixed correct set of primitives for a data structure. The set given above for $NatNum$ is small, but not minimal (we could get by without Add). We use a slight variation on the LSL notation to guide our choice of primitives. In the **introduces** section we specify primitive functions by labelled categories as described below.

1. Specify the **elements** of the domain — often this is an inductive definition:

   (a) Specify the **basis** element(s) — $Zero$ in the example above. (A function of no arguments is simply a constant.)

   (b) Specify how to generate compound elements of the domain. If it is an inductive definition, we show how to define new elements of the domain from existing elements — $Succ$ in the example above.

   This class of functions is known as the **constructors** for the data domain. Some domains may have more than one constructor function. (See the data structure *LogicWffs* below.)

(c) The *extremal clause*, which states that the only elements of the domain are those that can be generated from a finite number of applications of (a) and (b), is often assumed and not explicitly stated. We can make this explicit with an assertion that the data structure being defined is **generated by** the basis element(s) and constructor(s) provided ($Zero$ and $Succ$ in the example above). This guarantees that there are no "hidden" ways for an element to slip into the domain. If it cannot be generated by the methods given then it is not in the domain.

2. **Selectors** are primitives for decomposing compound elements of the domain — usually one considers providing selectors corresponding to the arguments of the constructor functions. In $NatNum$, since a natural number is *constructed* by applying $Succ$ to an existing natural number, we provide a selector $Pred$ that produces the natural number whose successor is a given number, thus, $Pred(Succ(x)) = x$.

   In defining binary trees in which a value is associated with each node one might provide a constructor, $MakeBtree(t1, v, t2)$, that builds a binary tree out of the given value v and two binary trees $t1$ and $t2$. One is then likely to want selectors that could decompose the resulting tree, selecting the value or the left or right subtree. So one is led to specify as primitives the selectors $Value(t), LeftSubTree(t)$ and $RightSubTree(t)$.

3. **Testers** are primitives that test elements of the domain — you may want to know if an element is a basis element, $IsZero$ in the example above, and you often want to be able to test the equality of elements of the domain (Eq).

4. In addition to the **constructors, selectors,** and **testers** mentioned above, you may wish to provide a set of primitives you expect to be necessary, or just useful, to anyone writing an application using the data structure. We label these primitives **utilities**.

When designing ADT's:

- We write our specifications in a representation independent way.
- Later we discuss possible representations for the elements of the data structure.
- Finally, we implement a data structure by writing programs that operate on a chosen representation in such a way that the values determined by the primitive functions can be computed while satisfying the equations of the trait.

1.1 A Familiar Example

Consider an example of an abstract data type with which you are familiar: $Stack$. Informally, a stack is a container that allows deposit and retrieval of items one at a time, and that maintains the order in which items are deposited so that the last one in is the first one out. (This is known as a "Last In First Out" or LIFO structure.)

In specifying the abstract data type $Stack$ we are concerned with the stack structure itself, not (necessarily) with the structure of the items in the stacks we generate. We assume that the item type is specified elsewhere, in a trait *ItemDef*, and that, at a minimum we can compare items for equality with a primitive Eq defined on the item type.

Note that the name of the trait, *StackofItems*, need not be the same as the name of the designated domain of elements, *Stack*, of the data structure. Again, in order to decide upon the primitives to be specified we consider the categories of constructors, selectors, and testers.

```
StackOfItems: trait
  includes ItemDef
  introduces
    elements basis MtStk : →Stack
              constructor Push: Item, Stack →Stack
    selectors Pop : Stack→Stack
              Top : Stack→Item
    testers IsEmpty : Stack→Boolean
            Equal : Stack, Stack→Boolean
  asserts
  Stack generated by MtStk, Push
  Stack partitioned by IsEmpty, Pop, Top
  Stack partitioned by Equal
  for all x, y : Item and l, m : Stack
    IsEmpty(MtStk) == true
    IsEmpty(Push(x, l)) == false
    Equal(MtStk, MtStk) == true
    Equal(MtStk, Push(x, l)) == false
    Equal(Push(x, l), MtStk) == false
    Equal(Push(x, l), Push(y, m)) == if Eq(x, y)
                                     then Equal(l, m)
                                     else false
    Top(Push(x, l)) == x
    Pop(Push(x, l)) == l
  exempting Top(MtStk), Pop(MtStk)
end StackOfItems
```

To support the modular design of specifications, LSL allows one to combine traits using **includes**. For example:

```
ThisTrait : trait
  includes ThatTrait, AnotherTrait
...
```

The clause **includes** *ThatTrait, AnotherTrait* within the definition of *ThisTrait*, can be considered shorthand for including all of the **introduces** and **asserts** clauses of *ThatTrait* and *AnotherTrait* in the body of *ThisTrait*. Thus by declaring **includes** *ItemDef* in the *StackofItems* trait, we have implicitly added all of the primitives and their constraints from *ItemDef* to *StackofItems*.

Another feature we hadn't seen before is the **partitioned by** clause. When we claim that *Stack* is "partitioned by" *IsEmpty*, *Pop*, and *Top*, we are saying that if you cannot tell the difference between two stacks by using these functions, then the stacks are equal. Clearly, if we have defined a function to test the equality of the elements of an ADT, then the ADT is **partitioned by** that function.

1.2 Enumerations and Unions

LSL provides shorthand for definitions of special kinds of traits that are frequently required. These include enumerations and unions. To specify a finite set of distinct constants and an operator to enumerate them, one uses the enumeration shorthand. For example,

TrafficLight **enumeration of** *green, yellow, red*

is equivalent to including a trait:

TrafficLightTrait **: trait**
introduces
  elements basis *green, yellow, red* : $\rightarrow$ *TrafficLight*
  utilities *succ* : *TrafficLight* $\rightarrow$ *TrafficLight*
asserts
  TrafficLight **generated by** *green, yellow, red*
  equations
    $green \neq yellow$
    $green \neq red$
    $yellow \neq red$
    $succ(green) == yellow$
    $succ(yellow) == red$
end *TrafficLightTrait*

Note that the enumeration shorthand does not define the *succ* of the last element in the enumeration to be the first element, so the *succ(red)* is left unspecified by the shorthand. Of course, one can always add such an equation in the **asserts** or **equations** part of a trait specification. (Note that while we had not discussed **equations** as a category before, the only difference between it and the axioms specified with equations before is that the equations listed here do not involve any variables — so it doesn't make sense to say "**for all** x : *TrafficLight*".)

The union shorthand allows one to succinctly define tagged union types, that is, unions in which the component from which an element came is discernible from an element of the union. For example, we will soon want to describe an ADT for formulas of Propositional Logic. The atomic formulas are either proposition letters (elements of *PropLetter*[1]) or *Boolean* constants. We can define *Atom* as a union of *PropLetter* and *Boolean* using the union shorthand.

Atom **union of** *pl:PropLetter, bconst:Boolean*

[1]*PropLetter*, standing for Proposition Letters, is the set of all uppercase letters of the alphabet.

The tags are the names provided before the component types; these tags are implicitly defined as unary functions mapping an element of the component type to an element of the union. The tags can also be used as a suffix to map from the union type to the component type. A unary function *tag* maps elements of the union type to the set of tags for that union. The "union shorthand" is then an abbreviation for including a trait that defines all of the above. For example, the union shorthand defining *Atom* is equivalent to including the following trait:

```
AtomDef: trait
    Atom_tag enumeration of pl, bconst
    introduces
        elements constructors pl: PropLetter → Atom
                              bconst: Boolean → Atom
        selectors __.pl: Atom → PropLetter
                  __.bconst: Atom → Boolean
        utilities tag: Atom → Atom_tag
    asserts
        Atom generated by pl, bconst
        Atom partitioned by .pl, .bconst, tag
        for all p : PropLetter, b : Boolean
            pl(p).pl == p
            bconst(b).bconst == b
            tag(pl(p)) == pl
            tag(bconst(b)) == bconst
end AtomDef
```

Using underbars "__" as argument placeholders we can specify infix and postfix functions. In the trait above, "*.pl*" and "*.bconst*" are postfix functions.

The **generated by** clause states that each element of *Atom* is produced by either *pl* or *bconst*. Like the extremal clause of an inductive definition, this declares that there are no elements of *Atom* other than those produced by the given set of functions. The **partitioned by** clause states that all distinct values of *Atom* can be distinguished by the given list of functions. That is, if it is impossible to get different results from these functions applied to two elements of *Atom*, then the elements are equal. This implicitly defines equality on elements of the data structure.

1.3 Specification of *LogicWffs*

Now we want to specify the data structure of well-formed formulas of Propositional Logic (*Wffs*), using a trait that we'll call *LogicWffs*. The elements of the domain for this data structure (*Wffs*) include the *Boolean* constants *true* and *false*, all proposition letters, and formulas that can be built from formulas using the connectives $\neg$, $\wedge$, and $\vee$.

```
LogicWffs: trait
    Atom union of pl:PropLetter, bconst:Boolean
```

introduces
  elements basis *mk_Wff*: $Atom \to Wffs$
    constructors $\neg$: $Wffs \to Wffs$
      $\_\_\vee\_\_ : Wffs,\ Wffs \to Wffs$
      $\_\_\wedge\_\_ : Wffs,\ Wffs \to Wffs$
  selectors *GetAtom*: $Wffs \to Atom$
    $FirstArg$: $Wffs \to Wffs$
    $SecondArg$: $Wffs \to Wffs$
  testers $IsAtomic$: $Wffs \to Boolean$
    $IsNeg$: $Wffs \to Boolean$
    $IsOr$: $Wffs \to Boolean$
    $IsAnd$: $Wffs \to Boolean$
asserts
Wffs **generated by** *MkWff*, $\neg$, $\vee$, $\wedge$
for all $x, y :$ *Wffs*, $z :$ *Atom*
  $GetAtom(mk\_Wff(z)) == z$
  $FirstArg(\neg(x)) == x$
  $FirstArg(x \vee y) == x$
  $FirstArg(x \wedge y) == x$
  $SecondArg(x \vee y) == y$
  $SecondArg(x \wedge y) == y$
  $IsAtomic(mk\_Wff(z)) == true$
  $IsAtomic(\neg(x)) == false$
  $IsAtomic(x \vee y) == false$
  $IsAtomic(x \wedge y) == false$
  $IsNeg(mk\_Wff(z)) == false$
  $IsNeg(\neg(x)) == true$
  $IsNeg(x \vee y) == false$
  $IsNeg(x \wedge y) == false$
  $IsOr(mk\_Wff(z)) == false$
  $IsOr(\neg(x)) == false$
  $IsOr(x \vee y) == true$
  $IsOr(x \wedge y) == false$
  $IsAnd(xmk\_Wff(z)) == false$
  $IsAnd(\neg(x)) == false$
  $IsAnd(x \vee y) == false$
  $IsAnd(x \wedge y) == true$
exempting for all $x,\ y :$ *Wffs*, $z :$ *Atom*
  GetAtom$(\neg(x))$, *GetAtom*$(x\wedge\ y)$,
  GetAtom$(x \vee\ y)$, *FirstArg(mk_Wff(z))*,
  SecondArg(mk_Wff(z)), *SecondArg*$(\neg(x))$
end *LogicWffs*

The first line following **trait** defines *Atom* as a tagged union of *PropLetter* and *Boolean.*

Note that in this case there are three different constructor functions that generate compound *Wffs* from simpler *Wffs*. There are also many basis elements of the data domain (*Atom*, made up of *Boolean* constants and all the proposition letters), thus, instead of a zero-ary function, representing an individual constant, we have a unary constructor that builds an atomic well-formed formula from an *Atom*. Thus we know that there are four types of elements of this domain, atomic elements and elements constructed using $\neg$, $\vee$, and $\wedge$. We can completely specify any function operating on this domain by specifying how it operates on each of the four possible kinds of elements. The functions defined on all four kinds of elements are defined over the entire domain, the **exempting** clause points out that we have not specified values for all four cases for the functions *FirstArg* and *SecondArg*.

1.4 LSL Summary

The abstract data type specifications we write in the Larch Shared Language require only a small subset of the language. Each trait is given a name. The name of the trait may or may not be used as the name of the designated domain of elements of the data structure being specified. It is possible to explicitly include information from other traits via the **include** declaration.

1.4.1 Generic Specifications

Another means of modularizing specifications is to define parameterized traits. For example, instead of using **include** *ItemDef* we can anticipate that the *StackofItems* trait will be useful for several different item types and parameterize the trait by the Item type.

StackofItems(Item) **: trait**

We might then refer to a trait specifying a stack of natural numbers as *StackofItems(NatNum)*. Since we don't always anticipate every way we may later want to parameterize a trait, we can also refer to a trait with an explicit renaming for any of the types and/or functions introduced by the trait. Thus, an unparameterized *StackofItems* **: trait** (without the **includes** *ItemDef* clause) can be referenced later as *StackofItems(NatNum* **for** *Item)*.

1.4.2 Primitives

The **introduces** section of a trait is used to specify the names and functionality of all primitives introduced by the trait. We categorize the primitives under the following headings:

elements This section is used to specify the elements of the ADT, often inductively by providing basis elements and constructors.

selectors These primitives select components of compound elements of the data structure. One may consider providing a selector for each argument in each constructor used to build elements.

testers These predicates might allow one to test for different types of elements (basis vs compound) or to check the elements of the structure (for equality or other properties).

utilities This is the most subjective category. A designer will include here those functions felt to be most desirable as primitives due to the anticipated needs of anyone using the data structure.

1.4.3 Axioms

The **asserts** section allows one to declare constraints that must be met by any implementation of the data structure. These constraints include axioms about the completeness of the specification of elements (**generated by**) and equivalence of elements (**partitioned by**), as well as axioms specifying equations that define relationships among elements of the ADT, and possibly other ADTs. The axioms often define the primitives (other than the constructors) inductively, specifying values for basis elements and then for constructed elements in terms of their components. The axioms need not completely define each primitive, as some applications may not make sense, or even represent errors. We can call attention to applications left unconstrained by using the **exempting** clause.

The syntax we use for LSL specifications of data structures can be summarized by the grammar in Figure 2. In the grammar, “::=” can be read “is a” or “can be replaced by.” Terminal symbols are in boldface (including the special symbols and punctuation $\wedge$, $\vee$, $\rightarrow$, =, ==,(,), ., ,, and :). A superscript of “*” indicates 0 or more occurrences, and a superscript “$^{+}$” indicates 1 or more occurrences. The superscript “$^{+}$” followed by a comma, indicates 1 or more occurrences, separated by commas. Optional items are enclosed in square brackets “[” and “]” and the set braces “{” and “}” are used to group items.

Figure 2: Grammar for enhanced subset of LSL

trait	::=	simpleId [**(**simpleId$^+$, **)**] **: trait**
		{ shorthand | external }* opPart* propPart* [exemptions]
shorthand	::=	enumeration | union
enumeration	::=	sort **enumeration of** simpleId$^+$,
union	::=	sort **union of** fields$^+$,
sort	::=	simpleId
fields	::=	simpleId$^+$, **:** sort
external	::=	**includes** traitRef$^+$,
traitRef	::=	{ simpleId | **(** simpleId$^+$, **)** } [**(** renaming **)**]
renaming	::=	replace$^+$, | name$^+$, { , replace }*
replace	::=	name **for** name
opPart	::=	**introduces** elements [selectors] [testers] [utilities]
elements	::=	[**elements** [**basis**]] opDcl$^+$ [**constructors** opDcl$^+$]
selectors	::=	**selectors** opDcl$^+$
testers	::=	**testers** opDcl$^+$
utilities	::=	**utilities** opDcl$^+$
opDcl	::=	name$^+$, **:** signature
signature	::=	sort*, $\rightarrow$ sort
propPart	::=	**asserts** genPartition* eqPart
genPartition	::=	sort { **generated** | **partitioned** **}** **by** operator$^+$,
operator	::=	name [**:** signature]
eqPart	::=	[**equations** equation$^+$] { **for all** varDcl$^+$, equation$^+$ }*
varDcl	::=	simpleId$^+$, **:** sort
equation	::=	term [== term]
term	::=	logicalTerm | **if** term **then** term **else** term
logicalTerm	::=	equalityTerm { logicalOp equalityTerm }*
logicalOp	::=	$\wedge$ | $\vee$ | $\rightarrow$ |=
equalityTerm	::=	simpleOpTerm [eqOp simpleOpTerm]
simpleOpTerm	::=	simpleOp$^+$ primary
		| primary simpleOp$^+$
		| primary { simpleOp primary }*
primary	::=	{ **(** term **)** | simpleId [**(** term$^+$**,** **)**] } { **.** simpleId | **:** sort }*
exemptions	::=	**exempting** [**for all** varDcl$^+$,] term$^+$,

1.5 Exercises

1. Provide a formal specification for the data structure *List*. Treat the elements of the domain of *List* as pure values. Assume that each element of the domain is a list of items of type *Item*, which is specified elsewhere. (Note the similarity between this structure and that defined for *Stack*.)

2. Write an LSL trait to specify the ADT *Queue*. A *Queue* is another structure that allows deposit and retrieval of one element at a time and maintains the order of its elements, but unlike a *Stack*, the elements of a *Queue* are retrieved in the same order in which they were deposited. Thus a *Queue* is a First In First Out (FIFO) structure.

3. Provide a formal specification for the data structure *Set*. Assume that each element of the domain is a set of items of type *Item*, which is specified elsewhere. (Since a set is an unordered collection, one must be careful not to rely on a specific order of insertion of elements when defining axioms that constrain the primitives.)

4. Provide a formal specification of the data structure *ArithExp*, of arithmetic expressions. An arithmetic expression can be a number, or a variable, or the sum or product of arithmetic expressions.

5. Provide a formal specification of the data structure *SymbolTable*, which is a structure that allows storing and retrieving of values associated with symbols. A *SymbolTable* can be empty, or it can contain bindings of symbols and values.

6. Provide a formal specification of the data structure *Environment*, which can be thought of as a stack of frames, each of which is a *SymbolTable*.

7. Give at least one advantage and at least one disadvantage of specifying a minimal number of primitive functions in a data structure specification.

2 Representation

Once we have specified the values and primitives of a data structure, as well as the constraints to be met by any implementation, we can consider alternatives for representing it. That is, we need to establish a convention for denoting elements of the data structure so there is no ambiguity about which element is intended. Given an instance of the representation there can be only one element of the data structure that is being denoted. (There is no complementary restriction that each element of the data structure have exactly one representation. For example, if we represent a set of elements as an unordered list of the elements, then several different lists may represent the same set.)

As an example, we will consider a few choices of representation for the ADT *StackOfItems*. In these notes we use Haskell as the programming language for implementation of ADTs. This constrains the syntax we use for expressing representations, but not the concepts described.

2.1 Representing Stacks as Lists

A Stack of elements of type `a` might conveniently be represented as a list, each of whose elements represents a single element of the stack. An empty list (or `[ ]` would represent an empty stack. In Haskell we could define such a representation by making the type (`Stack a`) a type synonym for the type `[a]`, as follows.

```
type Stack a = [a]
```

The benefit of defining the (`Stack a`) type as a type synonym is that we can use all the Haskell list functions predefined for us, as well as any additional list functions we have defined. Since `Stack` is just another name for a list type, all elements of type `Stack` can be treated as lists.

This describes the storage structure to be used for a stack, but it doesn't *implement* the stack. To accomplish the implementation we need to choose and implement primitive stack operations that provide all the functionality specified by the trait *StackOfItems*, but operate on the chosen representation. To ensure the correctness of our implementation we need to be satisfied that all the constraints supplied by the axioms in the **asserts** section of the trait are met. Before moving on to implementation, we consider an alternative representation.

2.2 Representing Stacks as a New Type

The drawback in representing stacks as lists is much the same as the benefit. We have many functions that operate on lists. Some of them make sense for stacks, but some do not. We might worry that someone writing a program that deals with stacks would be tempted to access and build stacks with the usual list functions rather than the stack operations provided. This could lead to stacks that no longer behave in their characteristic Last-In-First-Out way.

As an alternative to the type synonym definition, we consider a definition of a brand new type `Stack` with two constructors: `MtS` and `Push`

```
data Stack a = MtS | Push a (Stack a)
```

This representation provides some protection in that the only functions people can use to operate on things of type `Stack` will be functions specifically designed to work with stacks. Of course that means we are also constrained in the definition of primitive functions, we have to implement everything from scratch.

2.3 Representing Stacks as a New Type Using Lists

We noted that representing stacks as lists gave us the flexibility to use all the list functions in working with stacks. We also noted that it is dangerous to allow users of our stack module to treat stacks as lists, as they might end up working with stacks in ways that violate the specification. Happily, we have another alternative that provides us the flexibility we want in implementing the stack module, but hides the representation from the users of our module.

We define a new type `Stack` with a single constructor applied to a list of elements.

```
data Stack a = Stk [a]
```

As long as we export only the primitive functions mentioned in the specification, and NOT the constructor we defined above, then we can use the list functions within our implementation without making them accessible to someone using our module.

2.4 Exercises

1. Describe at least two representations for each of the ADTs specified in the exercises of section 1.5.

3 Implementation

Two major considerations when designing components for reuse are the flexibility of the component (range of variations it covers) and the degree of protection it provides. The flexibility is increased by making appropriate use of polymorphism in Haskell, providing much the same flexibility we gain by parameterizing our LSL specifications. Safety is provided in varying degrees by limiting access to the representation in defining the export list of the module.

To enhance the modularity of programs using our stack module, we prefer not to allow the user access to the representation of the stack. However, we do want to export the type `Stack` so they can define variables and functions using this type. (In the first representation above, as soon as we allow them the type `Stack`, since it is defined as a type synonym, they know the representation is as a list, and can use all list functions.)

Anyone using stacks should do so through a well-defined interface of primitive stack operations. The first impulse is to treat the **introduces** section of a trait specification as a rough outline of a module specification providing this interface, and to some extent it is just that. However, we still have the option of adding or deleting types or functions from the interface.

In providing an implementation of an ADT, we must be careful to satisfy the axioms of the trait specification. However, this does not constrain us to implement each primitive function as if the equations provided an operational specification. And, of course, the programs we write will operate on the representation of the ADT, not elements of the ADT itself.

Whenever we implement a data structure we should consider what *might* go wrong and decide how to deal with it. For example, the *StackOfItems* trait specification put no constraints on the values of *Top(MtStk)* and *Pop(MtStk).* Even though the trait leaves the behavior unspecified, an implementation cannot. In Haskell we can use the `error` function to signal that an anomalous call has been made and we can't proceed with the computation. (Haskell also has an exception handling mechanism, but for now we'll just use `error`.)

We are guided by the trait specification when defining the interface to our implementation, thus we must be sure to include primitives corresponding to those in the **introduces** section of the trait (*MtStk, Push, Pop, Top, IsEmpty,* and *Equal*). We give the export list at the top of the module, and provide the type signatures for all the functions before continuing with their implementations:

```
module StackOfItems (Stack, mtstk, push, pop,
                     top, isEmpty, isEqual)where

-- here are three versions, only one would appear in a given implementation
type Stack a = [a]
data Stack a = MtStk | Push a (Stack a)
data Stack a = Stk [a]

-- constructors (as far as the user knows)
mtstk ::  Stack a
push ::  a -> (Stack a) -> (Stack a)

-- selectors
pop ::  (Stack a) -> (Stack a)
top ::  (Stack a) -> a

-- testers
isEmpty ::  (Stack a) -> Bool
isEqual ::  (Eq a) => (Stack a) -> (Stack a) -> Bool

-- and then the implementation ...
```

Note the use of a *qualified type* in the signature for `isEqual`. In defining equality on two stacks we will need to be able to compare elements of the stacks for equality. The qualification "`(Eq a) =>`" says that "if type `a` is of class `Eq`, then the signature of `isEqual` is as follows. Knowing that `a` is of class `Eq`, we can assume that both "==" and "/=" are defined on elements of type `a`.

3.1 Haskell Implementation Type Synonym Stack

We saw how the *StackOfItems* trait could be parameterized by *Item*, the type of the items that might be pushed onto the stack. Haskell allows us similar flexibility in defining polymorphic types. All that the trait requires of the item type is the ability to check elements for equality. (This is needed in specifying equality of stacks.) Continuing the definition of the module started above, assuming the first definition of the type `Stack`:

```
⋮
type Stack a = [a]

⋮

mtstk = [ ]
push = (:)

pop [ ] = error "can't pop an empty stack"
pop (_:xs) = xs
top [ ] = error "can't get top of an empty stack"
top (x:_) = x

isEmpty [ ] = True
isEmpty (x:xs) = False
isEqual s1 s2 = s1 == s2
```

3.2 Haskell Implementation New Type Stack

Representing the stack as a synonym for a list allows us to use list functions in implementing the primitives. This makes our job easy, but presents a danger. If a user of our stack module is aware of this representation, they may decide to perform on stacks as lists, corrupting the representation in the process. For example, they could sort the elements of a stack, in which case it would no longer be a proper stack. We can force users of the module to treat stacks properly, using only the primitives we've defined along with other functions they define in terms of the primitives, by defining `Stack` as a new data type, and exporting only the primitive functions we want them to use in accessing the stack.

This has the dual advantage of protecting the integrity of the representation, and allowing us the flexibility of changing representation any time we wish without affecting any application that uses our stack module. As long as the interface stays the same, a client program will be unaffected. So, again assuming the same declaration part except for the type definition, we have:

```
⋮
data Stack a = MtStk | Push a (Stack a)
```

⋮

```
mtstk = MtStk
push x xs = Push x xs
-- note the difference between the function "push" and the constructor "Push"

pop MtStk = error "can't pop an empty stack"
pop (Push _ xs) = xs
top MtStk = error "can't get top of an empty stack"
top (Push x _) = x

isEmpty MtStk = True
isEmpty (Push x xs) = False
isEqual MtStk MtStk = True
isEqual (Push x xs) MtStk = False
isEqual MtStk (Push x xs) = False
isEqual (Push x xs) (Push y ys) = x==y && isEqual xs ys
```

This definition of the primitives looks much like the LSL specification. That is not always the case, but the case breakdown will look the same. We distinguish different forms of the structure by the way it is built with the constructors. This approach means a bit more work for us, since we cannot make use of previously defined functions as we did in the first example (with the type synonym.)

3.3 Haskell Implementation New Type Stack Using List

In order to get the best of both worlds, flexibility and ease of definition accorded by the type synonym, and safety provided by the new type definition, we consider a third option. In this case we defined `Stack` as a new type, but defined only a single constructor applied to a list of items. This way, we can protect the ADT from users while still making use of list functions in defining the primitives. The only additional complication is that we must strip the constructor before operating on the lists, and be sure to put it back when needing to produce a value of type `Stack`. So now our implementation looks as follows:

⋮

```
data Stack a = Stk [a]
```

⋮

```
mtstk = Stk [ ]
push x (Stk xs) = Stk (x:xs)

pop (Stk [ ]) = error "can't pop an empty stack"
pop (Stk (_:xs)) = (Stk xs)
```

```
top (Stk [ ]) = error "can't get top of an empty stack"
top (Stk (x:_)) = x

isEmpty (Stk [ ]) = True
isEmpty (Stk (x:xs)) = False
isEqual (Stk s1) (Stk s2) = s1 == s2
```

Thus we are able to make the definitions *almost* the same as we did for the type synonym, but we must still be careful about applying stack functions to stacks (`Stk` applied to a list) and list functions to lists (selected from the representation).

3.4 Correctness of the Implementation

The *StackOfItems* trait described stack values and constraints on operations involving them. We need to provide some way of mapping the constraints on the primitive operations of the trait to constraints on the functions of the package. We proceed by considering each equation and translating it into the language of Haskell. The equations, which were stated **for all** values of the variables used in stating them, become program constraints, which should also be understood to hold for all values of the variables used.

1. The equations constraining *IsEmpty*:

   $IsEmpty(MtStk) == \mathit{true}$
   $IsEmpty(Push(x, l)) == \mathit{false}$

   become program constraints:

   `isEmpty mtstk == True` for all values of `x` and `xs`.

   `IsEmpty (push x xs)== False` for all values of `x` and `xs`.

2. The equations constraining *Equal*:

   $Equal(MtStk, MtStk) == \mathit{true}$
   $Equal(MtStk, Push(x, l)) == \mathit{false}$
   $Equal(Push(x, l), MtStk) == \mathit{false}$
   $Equal(Push(x, l), Push(y, m)) ==$ **if** $Eq(x, y)$
   then $Equal(l, m)$
   else false

   become program constraints:

   `isEqual mtstk mtstk == True`
   `isEqual mtstk (push x xs) == False` for all values of `x` and `xs`.
   `isEqual (push x xs) mtstk == False` for all values of `x` and `xs`.
   `isEqual(push x xs) (push y ys)` ==

   again, for all values of `x, y, xs`, and `ys`.

3. The equation constraining *Top*:

   $Top(Push(x, l)) == x$

   becomes the program constraint:

   `top (push x xs) == x` for all values of `x` and `xs`.

4. The equation constraining *Pop*:

   $Pop(Push(x, l)) == l$

   becomes the program constraint:

   `pop (push x xs) == xs` for all values of `x` and `xs`.

We might also note that we have chosen to call error, with appropriate messages, in the two cases left unspecified by the trait, *Top*(*MtStk*), and *Pop*(*MtStk*).

Because Haskell is a functional language, we can actually prove most of the above statements about the Haskell program. However, we can also use these requirements to guide us in designing appropriate test programs. Clearly, where we've added the statement "for all values of `x` and `xs`," we cannot expect to actually test all possible values. We can choose appropriate sample values to test. For example, in checking the constraint on `top`, we should consider a case where `xs` is empty, and where it is not empty. We can use an "arbitrary" value for `x`, since its value has no affect on the constraint being checked.

When we test the cases for `isEqual` we have more work to do. We need to check enough cases to exercise every branch. We should consider checking two stacks that are equal (and empty, as well as non-empty, and possibly with a variety of number of elements), and stacks that are not equal. We should be sure to check unequal stacks that share some elements, and some of equal length as well as those with different lengths. It is often much more difficult to decide when you have checked "enough" different cases, than to prove the constraint for all cases (as described in SOE chapter 11).

Of course, for most programming languages in use today, you do not have such a well-behaved proof option. You are stuck with letting the constraints guide you in finding appropriate tests.

Regardless of the method used, a software engineer has a professional ethical obligation to ensure that an implementation meets its specification. Using a formal ADT specification helps guide such a task, whether used as basis for proof or simply as a guide to assist in generating test programs to check the constraints.

3.5 Exercises

1. Provide an implementation for one of the representations given in section 2.4. Discuss why you chose the representation you did.

2. Give a convincing argument that your implementation provided in the previous problem is correct with respect to its LSL specification.