The ABS Language Specification

February 11, 2011

Chapter 1

The ABS Language

This chapter gives a report on the core ABS language as it is implemented in the ABS tools. The ABS language is a class-based object-oriented language that features algebraic data types and side effect-free functions. Syntactically, the ABS language tries to be as close as possible to the Java language [?] so that programmers that are used to Java can easily use the ABS language without much learning effort.

1.1 Notation

In this chapter we often present the concrete syntax of the ABS language. To do so we use BNF[1] with the following denotations.

- `[x]` denotes zero or one occurrences of x.
- `{x}` denotes zero or more occurrences of x.
- `x | y` means one of either `x` or `y`
- [: x :] denotes the POSIX character class x.
- `'x'` denotes that `x` is a terminal symbol

1.2 Lexical Structure

This section describes the lexical structure of the ABS language. ABS programs are written in Unicode[2].

1.2.1 Line Terminators and White Spaces

Line terminators and white spaces are defined as in Java.

Syntax:

```
LineTerminator ::= \n | \r | \r\n
WhiteSpace     ::= LineTerminator | ' ' | \t | \f
```

[1]Backus-Naur Form
[2]`http://www.unicode.org`

1.2.2 Comments

Comments are pieces of text that are completely ignored and have no semantics in the ABS language. ABS supports two styles of comments: *end-of-line comments* and *traditional comments*.

End-Of-Line Comments

An end-of-line comment is a piece of text that starts with two slashes, e.g., `// text`. All text that follows `//` until the end of the line is treated as a comment.

Example:

```
// this is a comment
module A; // this is also a comment
```

Traditional Comments

A traditional comment is a piece of text that is enclosed in `/* */`, e.g., `/* this is a comment */`. Nested traditional comments are not possible.

Example:

```
/* this
is a multiline
comment */
```

1.2.3 Identifiers

ABS distinguishes *identifier* and *type identifier*. They differ in the first character, which must be a lower-case character for identifiers and an upper-case character for type identifiers.

Syntax:

```
Identifier    ::= [:lowercase:] {[:letter:] | [:digit:] | '_'}
TypeId        ::= [:uppercase:] {[:letter:] | [:digit:] | '_'}
```

1.2.4 Keywords

The following words are keywords in the ABS language and are *not* regarded as identifiers.

adds	after	assert	await	builtin	case
cog	core	class	data	def	delta
else	export	features	from	get	hasField
hasInterface	hasMethod	if	implements	import	in
interface	let	modifies	module	new	null
product	productline	removes	return	skip	suspend
this	type	when	while		

1.2.5 Literals

A *literal* is a textual representation of a value. ABS supports three kinds of literals, *integer literals*, *string literals*, and the *null literal*.

Syntax:

```
Literal       ::= IntLiteral
                | StringLiteral
                | NullLiteral
IntLiteral    ::= 0 | [1-9]{[0-9]}
StringLiteral ::= '"' {StringCharacter} '"'
NullLiteral   ::= 'null'
```

Where a `StringCharacter` is defined as in the Java language [**?**, p. 28]

1.2.6 Separators

The following characters are *separators*:

```
(   )   {   }   [   ]   ,   ;   :
```

1.2.7 Operators

The following tokens are *operators*:

```
||   &&   ==   !=   <   >   <=   >=   +   -   *   /   %   ~   &
```

1.3 Names and Types

1.3.1 Names

A *name* in ABS can either be a simple identifier as described above, or can be qualified with a type name, which represents a module.

Syntax:

```
TypeName     ::= TypeId {'.' TypeId}
Name         ::= Identifier | TypeName '.' Name
```

Examples for syntactically valid names are: `head`, `x`, `ABS.StdLib.tail`. Examples for type names are: `Unit`, `X`, `ABS.StdLib.Map`.

1.3.2 Types

Types in ABS are either plain type names or can have type arguments.

Syntax:

```
Type     ::= TypeName [TypeArgs]
TypeArgs ::= '<' TypeList '>'
TypeList ::= Type {',' Type}
```

Where TypeName can refer to a data type, an interface, a type synonym, and a type parameter. Note that classes cannot be used as types in ABS. In addition, only parametric data types can have type arguments. Examples for syntactically valid types are: Bool, ABS.StdLib.Int, List<Bool>, ABS.StdLib.Map<Int,Bool>.

1.4 Algebraic Data Types

Algebraic Data Types make it possible to describe data in an immutable way. In contrast to objects, data types do not have an identify and cannot be mutated. This makes reasoning about data types much simpler than about objects. Data types are built by using *Data Type Constructors* (or *constructors* for short), which describe the possible values of a data type.

Syntax:

```
DataTypeDecl   ::= 'data' TypeId [TypeParams] ['=' DataConstrList] ';'
TypeParams     ::= '<' TypeId {',' TypeId} '>'
DataConstrList ::= DataConstr {'|' DataConstr}
DataConstr     ::= TypeId ['(' [TypeList] ')']
```

Example:

```
data IntList = NoInt | Cons(Int, IntList);
data Bool = True | False;
```

1.4.1 Parametric Datatypes

Parametric Data Types are useful to define general-purpose datatypes, such as lists, sets or maps. Parametric datatypes are declared like normal datatypes but have an additional *type parameter* section inside broken brackets (< >) after the datatype name.

Example:

```
data List<A> = Nil | Cons(A, List<A>);
```

1.4.2 Predefined Datatypes

The following datatypes are predefined:

- data Bool = True | False;. The boolean type with constructors True and False and the usual Boolean infix and prefix operators.
- data Unit = Unit;. The unit type with only one constructor Unit (for methods without return values).
- data Int;. An arbitrary integer ($\mathbb{Z}$) for which values are constructed by using integer literals and arithmetic expressions.

- `data String;`. A string for which values are constructed by using string literals and operators.
- `data Fut<T>;`. Representing a future. A future cannot be explicitly constructed, but it is the result of an asynchronous method call. The value of a future an only be obtained by using the `get` expression (Sec. 1.7.4).
- `data List<A> = Nil | Cons(A, List<A>)`, with constructors `Nil` and `Cons(A, List<A>)`. This predefined datatype is used for implementing arbitrary n-ary constructors (see below).

1.4.3 N-ary Constructors

For datatypes of arbitrary size, like lists, maps and sets, it is undesirable having to write them down in the form of nested constructor expressions. For this purpose, ABS provides a special syntax for *n-ary constructors*, which are transformed into constructor expressions via a user-supplied function.

Example:

```
data Set<A> = EmptySet | Insert(A, Set<A>);
def Set<A> set<A>(List<A> l) =
   case l {
      Nil => EmptySet;
      Cons(hd, tl) => Insert(hd, set(tl));
   } ;

{
  Set<Int> s = set[1, 2, 3];
}
```

An expression *type[parameters*]* is transformed into a literal by handing it to a function named *type* which takes one parameter of type *List* and returns an expression of type *Type*. (It is desirable, although not currently enforced, that *type* and *Type* are the same word, just with different capitalization.)

1.4.4 Abstract Data Types

Using the module system (cf. Sec. 1.12) it is possible to define *abstract data types*. For an abstract data type, only the functions that operate on them are known to the client, but not its constructors. This can be easily realized in ABS by putting such a data type in its own module and by only exporting the data type and its functions, without exporting the constructors.

1.5 Functions

Functions in ABS define names for parametrized data expressions. A Function in ABS is always side effect-free, which means that it cannot manipulate the Heap.

Syntax:

```
FunctionDecl ::= 'def' 'Type' Identifier ['<' TypeIdList '>'] '(' ParamList ')' '='
                 FunBody ';'
FunBody      ::= 'builtin' | PureExp
TypeIdList   ::= TypeId {',' TypeId}
```

Example:

```
def Int length(IntList list) =
  case list {
    Nil => 0 ;
    Cons(n, ls) => 1 + length(ls) ;
  } ;
```

1.5.1 Parametric Functions

Parametric Functions serve to work with parametric datatypes in a general way. For example, given a list of any type, a parametric function `head` can return the first element, regardless of its type. Parametric functions are defined like normal functions but have an additional `type parameter` section inside broken brackets (< >) after the function name.

Example:

```
def A head<A>(List<A> list) =
  case list {
    Cons(x, xs) => x;
  } ;
```

1.6 Pure Expressions

Pure Expressions are side effect-free expressions. This means that these expressions cannot modify the Heap.

Syntax:

```
PureExp     ::= Variable
              | FieldAccess
              | ThisExp
              | NullLiteral
              | LetExp
              | DataConstrExp
              | FnAppExp
              | FnAppListExp
              | CaseExp
```

```
                | OperatorExp
                | '(' PureExp ')'
Variable    ::= Identifier
FieldAccess ::= ThisExp '.' Identifier
ThisExp     ::= 'this'
PureExpList ::= PureExp {',' PureExp}
```

1.6.1 Let Expressions

Let Expressions bind variable names to pure expressions.

Syntax:

```
LetExp ::= 'let' '(' Param ')' '=' PureExp 'in' PureExp
```

Example:

```
let (Bool x) = True in ~x
```

1.6.2 Datatype Constructor Expressions

Datatype Constructor Expressions are expressions that create data type values by using datatype constructors. Note that for datatype constructors that have no parameters the parentheses are optional.

Syntax:

```
DataConstrExp ::= TypeName
                | TypeName '(' [PureExpList] ')'
```

Example:

```
True
Cons(True, Nil)
ABS.StdLib.Nil
```

1.6.3 Function Applications

Function Applications apply functions to arguments.

Syntax:

```
FnAppExp ::= Name '(' [PureExpList] ')'
```

Example:

```
tail(Cons(True, Nil))
ABS.StdLib.head(list)
```

1.6.4 Case Expressions / Pattern Matching

ABS supports pattern matching by the *Case Expression*. It takes an expression as first argument, which a series of patterns is matched against. When a pattern matches, the corresponding expression on the right hand side is evaluated.

Syntax:

```
CaseExp       ::= 'case' PureExp '{' {CaseBranch} '}'
CaseBranch    ::= Pattern '=>' PureExp ';'
Pattern       ::= Identifier
                | Literal
                | ConstrPattern
                | '_'
ConstrPattern ::= TypeName ['(' [PatternList] ')']
PatternList   ::= Pattern {',' Pattern}
```

Patterns

There are five different kinds of patterns available in ABS:

- Pattern Variables (e.g., x, where x is not bound yet)
- Bound Variables (e.g., x, where x is bound)
- Literal Patterns (e.g., 5)
- Data Constructor Patterns (e.g., Cons(Nil,x))
- Underscore Pattern (_)

Pattern Variables. Pattern variables are simply unbound variables. Like the underscore pattern, these variables match every value, but, in addition, bind the variable to the matched value. The bound variable can then be used in the right-hand-side expression of the corresponding branch. Typically, pattern variables are used inside of data constructor patterns to extract values from data constructors. For example:

```
def A fromJust<A>(Maybe<A> a) =
  case a {
    Just(x) => x;
  } ;
```

Bound Variables. If a bound variable is used as a pattern, the pattern matches if the value of the case expression is equal to the value of the bound variable.

```
def Bool contains<A>(List<A> list, A value) =
  case list {
    Nil => False;
    Cons(value, _) => True;
    Cons(_, rest) => contains(rest, value);
  } ;
```

Literal Patterns. Literals can be used as patterns. This is similar to bound variables, because the pattern matches if the value of the case expression is equal to the literal value.

```
def Bool isEmpty(String s) =
  case b {
    "" => True;
    _  => False;
  } ;
```

Data Constructor Patterns. A data constructor pattern is like a standard data constructor expression, but where certain sub expressions can be patterns again.

```
def Bool negate(Bool b) =
  case b {
    True => False;
    False => True;
  } ;
```

```
def List<A> remainder(List<A> list) =
  case b {
    Cons(_, rest) => rest;
  } ;
```

Underscore Pattern. The underscore pattern (_) simply matches every value. It is in general used as the last pattern in a case expression to define a default case. For example:

```
def Bool isNil<A>(List<A> list) =
  case list {
    Nil => True;
    _ => False;
  } ;
```

Type Checking

A case expression is type-correct if and only if all its expressions and all its branches are type-correct and the right-hand side of all branches have a common super type. This common super type is also the type of the overall case expression.

A branch (a pattern and its expression) is type-correct if its pattern and its right-hand side expression are type-correct. A pattern is type-correct if it can match the corresponding case expression.

1.6.5 Operator Expressions

ABS has a number of predefined operators which can be used to form *Operator Expressions*.

Syntax:

```
OperatorExp ::= UnaryExp
              | BinaryExp
UnaryExp    ::= UnaryOp PureExp
UnaryOp     ::= '~' | '-'
BinaryExp   ::= PureExp BinaryOp PureExp
BinaryOp    ::= '==' | '!=' | '<' | '<=' | '>' | '>=' | '+' | '-' | '*' | / | %
```

Table 1.1 describes the meaning as well as the associativity and the precedence of the different operators. They are grouped according to precedence, as indicated by horizontal rules, from low precedence to high precedence.

1.7 Expression With Side Effects

Beside pure expressions, ABS has expressions with side effects. However, these expressions are defined in such a way that they can only have a single side effect. This means that subexpressions of expressions can only be pure expressions again. This restriction simplifies the reasoning about ABS expressions.

Syntax:

```
Exp    ::= PureExp
         | EffExp
EffExp ::= NewExp
         | SyncCall
         | AsyncCall
         | GetExp
```

1.7.1 New Expression

A *New Expression* creates a new object from a class name and a list of arguments. In ABS objects can be created in two different ways. Either they are created in the current COG, using the standard New Expressions, or they are created in a new COG by using the `new cog` expression.

Expression	Meaning	Associativity	Argument types	Result type
`e1 || e2`	logical or	left	Bool, Bool	Bool
`e1 && e2`	logical and	left	Bool, Bool	Bool
`e1 == e2`	equality	left	compatible	Bool
`e1 != e2`	inequality	left	compatible	Bool
`e1 < e2`	less than	left	Int, Int	Bool
`e1 <= e2`	less than or equal to	left	Int, Int	Bool
`e1 > e2`	greater than	left	Int, Int	Bool
`e1 >= e2`	greater than or equal to	left	Int, Int	Bool
`e1 + e2`	concatenation	left	String, String	String
`e1 + e2`	addition	left	Int, Int	Int
`e1 - e2`	subtraction	left	Int, Int	Int
`e1 * e2`	multiplication	left	Int, Int	Int
`e1 / e2`	division	left	Int, Int	Int
`e1 % e2`	modulo	left	Int, Int	Int
`~ e`	logical negation	right	Bool	Bool
`- e`	integer negation	right	Int	Int

Table 1.1: Operator expressions, grouped according to precedence from low to high.

Syntax:

```
NewExp ::= 'new' ['cog'] TypeName '(' PureExpList ')'
```

Example:

```
new Foo(5)
new cog Bar()
```

1.7.2 Synchronous Call Expression

A *Synchronous Call* consists of a target expression, a method name, and a list of argument expressions.

Syntax:

```
SyncCall ::= PureExp '.' Identifier '(' PureExpList ')'
```

Example:

```
Bool b = x.m(5);
```

1.7.3 Asynchronous Call Expression

An *Asynchronous Call* consists of a target expression, a method name, and a list of argument expressions. Instead of directly invoking the method, an asynchronous method call creates a new *Task* in the target COG, which is executed asynchronously.

Syntax:

```
AsyncCall ::= PureExp '!' Identifier '(' PureExpList ')'
```

Example:

```
Fut<Bool> f = x!m(5);
```

1.7.4 Get Expression

A *Get Expression* is used to obtain the value from a future. The current task is blocked until the value of the future is available, i.e., until the future has been resolved. No other task in the COG can be activated in the meantime.

Syntax:

```
GetExp ::= PureExp '.' 'get'
```

Example:

```
Bool b = f.get;
```

1.8 Statements

In contrast to expressions, *Statements* in ABS are not evaluated to a value. If one wants to assign a value to statements it would be the `Unit` value.

Syntax:

```
Statement    ::= CompoundStmt
               | VarDeclStmt
               | AssignStmt
               | AwaitStmt
```

```
                 | SuspendStmt
                 | SkipStmt
                 | AssertStmt
                 | ReturnStmt
                 | ExpStmt
CompoundStmt ::= Block
                 | IfStmt
                 | WhileStmt
```

1.8.1 Block

A block consists of a sequence of statements and defines a name scope for variables.

Syntax:

```
Block ::= '{' {Statement} '}'
```

1.8.2 If Statement

Syntax:

```
IfStmt ::= 'if' '(' PureExp ')' Stmt ['else' Stmt]
```

Example:

```
if (5 < x) {
  y = 6;
} else {
  y = 7;
}

if (True)
  x = 5;
```

1.8.3 While Statement

Syntax:

```
'while' '(' PureExp ')' Stmt
```

Example:

```
while (x < 5)
   x = x + 1;
```

1.8.4 Variable Declaration Statements

A variable declaration statement is used to declare variables.

Syntax:

```
VarDeclStmt ::= TypeName Identifier ['=' Exp] ';'
```

A variable has an optional *initialization expression* for defining the initial value of the variable. The initialization expression is *mandatory* for variables of data types. It can be left out only for variables of reference types, in which case the variable is initialized with `null`.

Example:

```
Bool b = True;
```

1.8.5 Assign Statement

The *Assign Statement* assigns a value to a variable or a field.

Syntax:

```
AssignStmt ::= Variable '=' PureExp ';'
             | FieldAccess '=' PureExp ';'
```

Example:

```
this.f = True;
x = 5;
```

1.8.6 Await Statement

Await Statements suspend the current task until the given *Guard* is true.

Syntax:

```
AwaitStmt  ::= 'await' Guard ';'
Guard      ::= ClaimGuard
             | PureExp
             | Guard '&' Guard
```

```
ClaimGuard ::= Variable '?'
             | FieldAccess '?'
```

Example:

```
Fut<Bool> f = x!m();
await f?;
await this.x == True;
await f? & this.y > 5;
```

1.8.7 Suspend Statement

A *Suspend Statement* is just syntactic sugar for an Await Statement with a `True` guard.

Syntax:

```
SuspendStmt ::= 'suspend' ';'
```

Example:

```
suspend ;
```

1.8.8 Skip Statement

The *Skip Statement* is a statement that does nothing.

Syntax:

```
SkipStmt ::= 'skip' ';'
```

1.8.9 Assert Statement

An *Assert Statement* is a statement for asserting certain conditions.

Syntax:

```
AssertStmt ::= 'assert' PureExp ';'
```

Example:

```
assert x != null;
```

1.8.10 Return Statement

A *Return Statement* defines the return value of a method. A return statement can only appear as a last statement in a method body.

Syntax:

```
ReturnStmt ::= 'return' PureExp ';'
```

Example:

```
return x;
```

1.8.11 Expression Statement

An *Expression Statement* is a statement that only consists of a single expression. Such statements are only executed for the effect of the expression.

Syntax:

```
ExpStmt ::= Exp ';'
```

Example:

```
new C(x);
```

1.9 Classes and Interfaces

Objects in ABS are built from *classes*, which implement *interfaces*. Only interfaces can be used as types in ABS.

1.9.1 Interfaces

Interfaces in ABS are similar to interfaces in Java. They have a name, which defines a nominal type, and they can *extend* arbitrary many other interfaces. The interface body consists of a list of method signature declarations. Method names start with a lowercase letter.

Syntax:

```
InterfaceDecl ::= 'interface' TypeId ['extends' TypeName {',' TypeName}] '{' MethSigList '}
MethSigList   ::= [MethSig {',' MethSig}]
MethSig       ::= Type Identifier '(' ParamList ')' ';'
ParamList     ::= [Param {',' Param}]
Param         ::= Type Identifier
```

Example:

```
interface Foo {
   Unit m(Bool b, Int i);
}

interface Bar extends Foo {
   Bool n();
}
```

1.9.2 Classes

Like in typical class-based languages, classes in ABS are used to create objects. Classes can implement an arbitrary number of interfaces. ABS does not support inheritance, as code reuse in ABS is realized by delta modules. Classes do not have constructors in ABS but instead have *class parameters* and an optional *init block*. Class parameters actually define additional fields of the class that can be used like any other declared field.

Syntax:

```
ClassDecl     ::= 'class' TypeId ['(' ParamList ')'] ['implements' TypeName {',' TypeName}]
                  '{' FieldDeclList [Block] MethDeclList '}'
FieldDeclList ::= FieldDecl {',' FieldDecl}
FieldDecl     ::= TypeId Identifier ['=' DataExp] ';'
MethDeclList  ::= MethDecl {',' MethDecl}
MethDecl      ::= Type Identifier '(' ParamList ')' Block
```

Example:

```
class Foo(Bool b, Int i) implements Bar, Baz {
   Int j = 5;
   Bar b;

   {
     j = i;
   }

   Bool m() {
     return True;
   }
}
```

Active Classes

A class can be *active* or *passive*. Active classes start an activity "on their own" upon creation, passive classes only react to incoming method calls.

A class is active if and only if it has a *run method*:

Example:

```
Unit run() {
  // active behavior ...
}
```

The run method is called after object initialization.

1.10 Annotations

ABS supports *Annotations* to enrich an ABS model with additional information, for example, to realize pluggable type systems. Annotations can appear before any declaration and type usage in ABS programs (which is not given in the grammar definitions, to improve readability).

Syntax:

```
Annotation ::= '[' [TypeName ':'] PureExp ']'
```

Example:

```
[LocationType:Near] Peer p;
[Far] Network n;
List<[Near] Peer> peers = Nil;
```

1.10.1 Type Annotations

ABS has a predefined “meta-annotation” `TypeAnnotation` to declare annotations to be *Type Annotations*. Data types that are annotated with that annotation are specially treated by the ABS compiler to support an easier implementation of pluggable type systems.

Example:

```
[TypeAnnotation]
data LocationType = Far | Near | Somewhere | Infer;
```

1.11 Type Synonyms

Type Synonyms define synonyms for otherwise defined types. Type synonyms start with an upper-case letter.

Syntax:

```
TypeSynDecl ::= TypeId '=' TypeName ';'
```

Example:

```
type Filename = String ;
type IntList = List<Int> ;
```

1.12 Modules

For name spacing, code structuring, and code hiding purposes, ABS offers a module system. The module system of ABS is very similar to that of Haskell [?]. It uses, however, a different syntax that is similar to that of Java [?] and Python.

Syntax:

```
ModuleDecl  ::= 'module' TypeName ';' [ExportList] [ImportList] {Decl} [Block]
ExportList  ::= Export {',' Export}
ImportList  ::= Import {',' Import}
Export      ::= 'export' AnyNameList ['from' TypeName] ';'
              | 'export' '*' ['from' TypeName] ';'
Import      ::= 'import' AnyNameList ['from' TypeName] ';'
              | 'import' '*' 'from' TypeName ';'
AnyNameList ::= AnyName [',' AnyName]
AnyName     ::= Name | TypeName
Decl        ::= FunDecl | TypeSynDecl | DataTypeDecl |
                InterfaceDecl | ClassDecl | DeltaDecl
```

A module with name `MyModule` is declared by writing

```
module MyModule ;
```

This declaration introduces a new module name `MyModule` which can be used to qualify names. All declarations which follow this statement belong to the module `MyModule`. A module name is a type name and must always start with an upper case letter.

1.12.1 Exporting

By default, modules do not export any names. In order to make names of a module usable to other modules, the names have to be *exported*. Exporting is done by writing one or several *exports* after the module declaration. For example, to export a data type and a data constructor, one can write something like this:

```
module Drinks ;
export Drink, Milk;
data Drink = Milk | Water;
```

Note that in this example, the data constructor `Water` is not exported, and can thus not be used by other modules. By only exporting the data type without any of its constructors one can realize *abstract data types*.

Exporting Everything

Sometimes one wants to export everything from a module. In that case one can write:

```
export * ;
```

In this case, all names that are *defined* in the module are exported, in particular, this means that imported names are *not* exported.

1.12.2 Importing

In order to use exported names of a module in another module, the names have to be *imported*. After the list of export statements follows an optional list of *imports*, which are used to import names from other modules. For example, to write a module that imports the `Drink` data type of the module `Drinks` one can write something like this:

```
module Bar;
import Drinks.Drink;
```

After a name has been imported, it can be used inside the module in a fully qualified way.

Unqualified Importing

To use a name from another module in an unqualified way one has to use *unqualified imports*. For example, to use the `Milk` data constructor inside the `Bar` module, without having to qualify it with the `Drinks` module each time, one uses the unqualified import statement:

```
module Bar;
import Drinks.Drink;
import Milk from Drinks;
```

Note that this kind of import also imports the qualified names. So in this example the names `Milk` and `Drinks.Milk` can be used inside the module `Bar`.

To use all exported names from another module in an unqualified way one can write:

```
import * from SomeModule;
```

1.12.3 Exporting Imported Names

It is possible to export names that have been imported. For example,

```
module Bar;
export Drink;
import * from Drinks;
```

exports data type `Drink` that has been imported from `Drinks`

To export all names imported from a certain module one can write

```
export * from SomeModule ;
```

In this case, all names that have been imported from module `SomeModule` are exported. For example,

```
module Bar;
export * from Drinks;
import * from Drinks;
```

exports all names that are exported by module `Drinks`.

However, in this example:

```
module Bar;
export * from Drinks;
import Drink from Drinks;
```

only Drink is exported as this is the only name imported from module Drinks. Note: only names that are visible in a module can be exported by that module.

To only export some names from a certain module one can write, for example:

```
module Bar;
export Drink from Drinks;
import * from Drinks;
```

only exports Drink from module Drinks

1.13 Model

A *Model* in ABS represents a type-closed set of *Modules*. A Module defines a set of declarations and an optional *Main Block*. Modules reside in *Compilation Units*, which are typically represented by files ending with .abs. A Model is thus set of Compilation Units.

Syntax:

```
Model           ::= {CompilationUnit}
CompilationUnit ::= {ModuleDecl}
```

Appendix A

ABS Standard Library

```
module ABS.StdLib;
export *;

data Unit = Unit;                // builtin
data String;                     // builtin
data Int;                        // builtin
data Bool = True | False;        // builtin
data Fut<A>;                     // builtin

def Bool and(Bool a, Bool b) = a && b;
def Bool not(Bool a) = ~a;

def Int max(Int a, Int b) =
    case a > b { True => a; False => b; };

def Int abs(Int x) =
    case x > 0 { True => x; False => -x; };

data Maybe<A> = Nothing | Just(A);

def A fromJust<A>(Maybe<A> a) = case a { Just(j) => j; };
def Bool isJust<A>(Maybe<A> a) =
    case a { Just(j) => True; Nothing => False; };

data Either<A, B> = Left(A) | Right(B);

def A left<A,B>(Either<A, B> val) =
    case val { Left(x) => x; };

def B right<A,B>(Either<A, B> val) =
    case val { Right(x) => x; };

def Bool isLeft<A,B>(Either<A, B> val) =
    case val { Left(x) => True; _ => False; };

def Bool isRight<A,B>(Either<A, B> val) = ~isLeft(val);


data Pair<A, B> = Pair(A, B);

def A fst<A, B>(Pair<A, B> p) = case p { Pair(s, f) => s; };
def B snd<A, B>(Pair<A, B> p) = case p { Pair(s, f) => f; };


data Triple<A, B, C> = Triple(A, B, C);

def A fstT<A, B, C>(Triple<A, B, C> p) =
    case p { Triple(s, f, g) => s; };

def B sndT<A, B, C>(Triple<A, B, C> p) =
    case p { Triple(s, f, g) => f; };

def C trd<A, B, C>(Triple<A, B, C> p) =
    case p { Triple(s, f, g) => g; };

// Sets
data Set<A> = EmptySet | Insert(A, Set<A>);

// set constructor helper
def Set<A> set<A>(List<A> l) =
    case l {
       Nil => EmptySet;
       Cons(x,xs) => Insert(x,set(xs));
    };

/**
 * Returns True if set 'ss' contains element 'e', False otherwise.
 */
def Bool contains<A>(Set<A> ss, A e) =
  case ss {
    EmptySet => False ;
    Insert(e, _) => True;
    Insert(_, xs) => contains(xs, e);
  };

/**
 * Returns True if set 'xs' is empty, False  otherwise.
 */
def Bool emptySet<A>(Set<A> xs) = (xs == EmptySet);
```

```
/**
 * Returns the size of set 'xs'.
 */
def Int size<A>(Set<A> xs) =
   case xs {
      EmptySet => 0 ;
      Insert(s, ss) => 1 + size(ss);
   };

def Set<A> union<A>(Set<A> set1, Set<A> set2) =
   case set1 {
      EmptySet => set2;
      Insert(a, s) => union(s,insertElement(set2,a));
   };

/**
 * Returns a set with all elements of set 'xs' plus element 'e'.
 * Returns 'xs' if 'xs' already containts 'e'.
 */
def Set<A> insertElement<A>(Set<A> xs, A e) =
  case contains(xs, e) {
    True => xs;
    False => Insert(e, xs);
  };

/**
 * Returns a set with all elements of set 'xs' except element 'e'.
 */
def Set<A> remove<A>(Set<A> xs, A e) =
  case xs {
     EmptySet => EmptySet ;
     Insert(e, ss) => ss;
     Insert(s, ss) => Insert(s,remove(ss,e));
  };

// checks whether the input set has more elements to be iterated.
def Bool hasNext<A>(Set<A> s) = ~ emptySet(s);

// Partial function to iterate over a set.
def Pair<Set<A>,A> next<A>(Set<A> s) =
   case s {
      Insert(e, set2) => Pair(set2,e);
   };

// Lists
```

```
data List<A> = Nil | Cons(A, List<A>);

def List<A> list<A>(List<A> l) = l; // list constructor helper

/**
 * Returns the length of list 'list'.
 */
def Int length<A>(List<A> list) =
   case list {
      Nil => 0 ;
      Cons(p, l) => 1 + length(l) ;
   };

/**
 * Returns True if list 'list' is empty, False otherwise.
 */
def Bool isEmpty<A>(List<A> list) = list == Nil;

/**
 * Returns the first element of list 'list'.
 */
def A head<A>(List<A> list) =
   case list { Cons(p,l) => p ; };

/**
 * Returns a (possibly empty) list containing all elements of 'list'
 * except the first one.
 */
def List<A> tail<A>(List<A> list) =
   case list { Cons(p,l) => l ; };

/**
 * Returns element 'n' of list 'list'.
 */
def A nth<A>(List<A> list, Int n) =
  case n {
    0 => head(list) ;
    _ => nth(tail(list), n-1);
  };

/**
 * Returns a list where all occurrences of a have been removed
 */
def List<A> without<A>(List<A> list, A a) =
  case list {
     Nil => Nil;
     Cons(a, tail) => without(tail,a);
     Cons(x, tail) => Cons(x, without(tail,a));
  };

/**
 * Returns a list containing all elements of list 'list1'
 * followed by all elements of list 'list2'.
 */
def List<A> concatenate<A>(List<A> list1, List<A> list2) =
  case list1 {
    Nil => list2 ;
    Cons(head, tail) =>  Cons(head, concatenate(tail, list2));
  };

/**
 * Returns a list containing all elements of list 'list' followed by 'p'.
 */
def List<A> appendright<A>(List<A> list, A p) =
    concatenate(list, Cons(p, Nil));

/**
 * Returns a list containing all elements of 'list' in reverse order.
 */
def List<A> reverse<A>(List<A> list) =
  case list {
     Cons(hd, tl) => appendright(reverse(tl), hd);
     Nil => Nil;
  };

/**
 * Returns a list of length 'n' containing 'p' n times.
 */
def List<A> copy<A>(A p, Int n) =
   case n { 0 => Nil; m => Cons(p,copy(p,m-1)); };




// Maps
data Map<A, B> = EmptyMap | InsertAssoc(Pair<A, B>, Map<A, B>);
 // map constructor helper (does not preserve injectivity)
def Map<A, B> map<A, B>(List<Pair<A, B>> l) =
  case l {
     Nil => EmptyMap;
     Cons(hd, tl) => InsertAssoc(hd, map(tl));
  };
```

```
def Map<A, B> removeKey<A, B>(Map<A, B> map, A key) = // remove from the map
  case map {
    InsertAssoc(Pair(key, _), map) => map;
    InsertAssoc(pair, tail) => InsertAssoc(pair, removeKey(tail, key));
  };


def List<B> values<A, B>(Map<A, B> map) =
  case map {
    EmptyMap => Nil ;
    InsertAssoc(Pair(_, elem), tail) => Cons(elem, values(tail)) ;
  };

/**
 * Returns a set containing all keys of map 'map'.
 */
def Set<A> keys<A, B>(Map<A, B> map) =
  case map {
    EmptyMap => EmptySet ;
    InsertAssoc(Pair(a, _), tail) => Insert(a, keys(tail));
  };

/**
 * Returns the value associated with key 'k' in map 'ms'.
 */
def B lookup<A, B>(Map<A, B> ms, A k) = // retrieve from the map
  case ms {
     InsertAssoc(Pair(k, y), _) => y;
     InsertAssoc(_, tm) => lookup(tm, k);
  };

/**
 * Returns the value associated with key 'k' in map 'ms', or the value 'd'
 * if 'k' has no entry in 'ms'.
 */
def B lookupDefault<A, B>(Map<A, B> ms, A k, B d) = // retrieve from the map
  case ms {
     InsertAssoc(Pair(k, y), _) => y;
     InsertAssoc(_, tm) => lookupDefault(tm, k, d);
     EmptyMap => d;
  };

/**
 * Returns a map with all entries of 'map' plus an entry 'p',
 * which might override but not remove another entry with the same key.
 */
def Map<A, B> insert<A, B>(Map<A, B> map, Pair<A, B> p) = InsertAssoc(p, map);

/**
 * Returns a map with all entries of 'ms' plus an entry mapping 'k' to 'v',
 * minus the first entry already mapping 'k' to a value.
 */
def Map<A, B> put<A, B>(Map<A, B> ms, A k, B v) =
  case ms {
    EmptyMap => InsertAssoc(Pair(k, v),EmptyMap);
    InsertAssoc(Pair(k, _), ts) => InsertAssoc(Pair(k, v), ts);
    InsertAssoc(p, ts) => InsertAssoc(p, put(ts, k, v));
  };

/**
 * Returns a string with the base-10 textual representation of 'n'.
 */
def String intToString(Int n) =
  case n < 0 {
    True => "-" + intToStringPos(-n);
    False => intToStringPos(n);
  };

def String intToStringPos(Int n) =
  let (Int div) = (n / 10) in
  let (Int res) = (n % 10) in
  case n {
    0 => "0"; 1 => "1"; 2 => "2"; 3 => "3"; 4 => "4";
    5 => "5"; 6 => "6"; 7 => "7"; 8 => "8"; 9 => "9";
    _ => intToStringPos(div) + intToStringPos(res);
  };

/**
 * Returns a substring of string str of the given length starting from start (inclusive)
 * Where the first character has index 0
 *
 * Example:
 *    substr("abcde",1,3) => "bcd"
 *
 */
def String substr(String str, Int start, Int length) = builtin;

/**
 * Returns the length of the given string
 */
```

```
def Int strlen(String str) = builtin;

// A Time datatype.
data Time = Time(Int);
def Int currentms() = builtin;
def Time now() = Time(currentms());
def Int timeval(Time t) = case t { Time(v) => v; };
// use this like so:
//   Time t = now(); await timeDifference(now(), t) > 5;
def Int timeDifference(Time t1, Time t2) =
  abs(timeval(t2) - timeval(t1));

/**
 * Annotation data type to define the type of annotations
 * currently only TypeAnnotation exists
 */
data Annotation = TypeAnnotation;

[TypeAnnotation]
data LocationType = Far | Near | Somewhere | Infer;

/**
 * Can be used to annotated classes and to ensure that
 * classes are always instantiated in the right way.
 * I.e. classes annotated with [COG] must be created by using
 * new cog, class annotated with [Plain] must be created by using
 * just new, without cog.
 */
data ClassKindAnnotation = COG | Plain;

/**
 * Declare local variables to be final
 */
data FinalAnnotation = Final;

/**
 * Declare methods to be atomic, i.e., such methods must not
 * contain scheduling code and also no .get
 */
data AtomicityAnnotation = Atomic;
```