CAL User's Guide

Contributors: Magnus Byne, Bo Ilic, Edward Lam, Joseph Wong, James Wright
Last modified: August 24, 2007

Contents

1 Overview

This is intended to be the primary document describing the use of the CAL language. It contains both introductory material and reference material.

CAL is a lazy functional language influenced by Haskell. It has an expression-based syntax akin to other formula languages (such as those found in Excel and Crystal Reports). It also features a powerful and flexible type system which allows the compiler to perform precise compile-time checking of function calls. In addition, it has a simple syntax for accessing Java objects, methods, and fields from within CAL.

Lazy evaluation means that the values of expressions are computed only when they are required by other expressions. This increases efficiency, as unnecessary calculations are avoided. It also allows for the definition of infinite data structures (such as the list of all even integers, for example), since only those parts of the data structure that are used will be computed. This makes it practical to divide a program into a generator that constructs many possible answers and a selector which chooses the appropriate ones.

1.1 Document layout

The User's Guide is divided into four main sections: Getting Started with CAL, the Language Reference, the Standard Library Reference, and the Appendices. Getting Started with CAL provides a brief introduction to new users of CAL. The Language Reference documents the features of the CAL language. The Standard Library Reference documents CAL's standard library modules (which corresponds fairly closely to the standard library of most object-oriented languages).

Finally, the Appendices section provides supplementary reference information: some suggested further reading and a reference list of CAL keywords.

1.2 Call for feedback

This document aims to be as complete and easy-to-use as possible. Any feedback you might have to help improve it would be very welcome. Please send any comments, suggestions, or questions to the CAL Language Discussion forum on Google Groups (http://groups.google.com/group/cal_language).

2 Getting Started with CAL

This section aims to give an introduction to the basic features of CAL. You are encouraged to follow the examples as you progress through the section. Some concepts may be used before they are introduced.

2.1 The ICE environment

ICE is the Interactive CAL Environment. It is a program that allows you to load CAL modules, and then type in expressions to be evaluated in the context of those modules. You may find it helpful to load the example definitions into ICE and evaluate some expressions against them.

2.1.1 Running ICE

To launch ICE, simply run the included `ICE.bat` batch file on Windows, or the `ICE.sh` shell script on UNIX/Linux platforms.

2.1.2 Using ICE to run the example code

It is only possible to enter expressions at the ICE prompt. Function definitions, type declarations, and other declarations must appear in modules. The best way to enter these declarations is to create your own module file. The declarations can be typed into the module file, which is then reloaded into the ICE environment.

Here are the steps that could be used in an ICE session to try some example code:

1. Use a text editor to create a file `UserGuideExamples.cal` in the "samples/simple/`CAL`" folder of your Quark distribution.
2. Use the text editor to paste the following code into the new file:
```
module UserGuideExamples;
import Cal.Core.Prelude using
    typeConstructor = Int, Double, String, Boolean, Char,
                      Integer, JObject, JList, Maybe, Ordering;
    dataConstructor = False, True, LT, EQ, GT, Nothing, Just;
    typeClass = Enum, Eq, Ord, Num, Inputable, Outputable;
    function =
        add, append, compare,
        concat, const, doubleToString, equals,
        error, fromJust, fst, input, intToString, isNothing,
        isEmpty, max, mod, not, output, round, seq, snd,
        toDouble, field1, field2, field3, upFrom, upFromTo;
    ;
import Cal.Collections.List using
    function =
        all, chop, filter, foldLeft, foldLeftStrict, foldRight,
        head, intersperse, last, list2, map, product, reverse,
        subscript, sum, tail, take, zip, zip3, zipWith;
    ;
import Cal.Collections.Array;
import Cal.Core.Bits;
```

```
import Cal.Core.Debug;
import Cal.Core.Dynamic using
    typeConstructor = Dynamic;
    function = fromDynamic, fromDynamicWithDefault, toDynamic;
    ;
import Cal.Utilities.Math using
    function = truncate;
    ;
import Cal.Utilities.StringNoCase;
import Cal.Core.String;
```

3. Save `UserGuideExamples.cal`
4. Use a text editor to add the following line to the end of the "`samples/simple/Workspace Declaration/cal.samples.cws`" file:

   ```
   StandardVault UserGuideExamples
   ```

5. Run `ICE.bat` (or `ICE.sh`).
6. Set the current module to be `UserGuideExamples` by entering the following at the ICE prompt:

   ```
   :sm UserGuideExamples
   ```

7. In a text editor, add some function definitions to the end of `UserGuideExamples.cal` and save the file. For example:

   ```
   myFactorial :: Integer -> Integer;
   myFactorial n =
       product (upFromTo 1 n);
   ```

8. Use the `:rc` command from the ICE prompt to recompile and load all changed module files:

   ```
   :rc
   ```

9. Enter some expressions at the ICE prompt for evaluation:

   ```
   myFactorial 5
   ```

The edit-recompile-evaluate cycle can be repeated any number of times.

2.2 Values and types

In CAL, all computations are done by evaluating expressions. An expression is an operation either on values, or upon other expressions. The most familiar type of expression is the arithmetic expression:

```
1.2 + 5.0
returns 6.2
```

The example expression above consists of the addition operation (+) applied to two subexpressions. The subexpressions are the value 1.2 and the value 5.0. The expression evaluates to the `Double` value 6.2.

Every valid CAL expression evaluates to a value. This document gives the value of example expressions after the word ***returns*** in bold italics (as in the above example, which has the value 6.2). The text after and including ***returns*** is not part of the expression itself.

Every value in CAL has a type. Intuitively, a type is a set of all the values that are a part of that type. Types can have sizes ranging from a single value (e.g. the `()` type), to an infinite number of values (e.g. the `Integer` type).

Try typing a few arithmetic expressions into ICE's command line. Ex:

```
5.0 * 10.0 / 2.5
returns 20.0

"One string " ++ "Another string"
returns "One string Another string"
```

2.2.1 Type inference

The CAL compiler uses a process known as type inference to determine the type of each expression that it evaluates. Type inference determines the type of the expression's result based upon the types of the operations and values that make up the expression.

```
1.2 + 5.0
returns 6.2
```

To return to our simple arithmetic example, both `1.2` and `5.0` are of type `Double` (i.e., double-precision floating point), and `+` is an operation of type `Num a => a -> a -> a` (i.e., a function that takes two arguments of some numeric type `a`, and returns a result of the same numeric type). Based upon this information, the inferencer is able to determine that the type of the result must be `Double`. We see from the actual result (`6.2`) that this is indeed the case.

ICE provides a command (`:t`) that allows you to inspect the type that the compiler has inferred for a given expression:

```
:t "One string " ++ " another"
outputs String

:t 1.2 + 5.0
outputs Double
```

We give the result as an ***outputs*** rather than as a ***returned*** value, because the above are commands to the ICE environment, not expressions.

CAL is a strongly-typed language. This means that the compiler checks each expression to ensure that it has a consistent type. If the inferencer cannot determine a consistent type for an application, compilation will fail with an error:

```
1.2 + "hello"
Error: Type error applying the operator "+" to its second
argument. Caused by: Type clash: type constructor Prelude.Double
does not match Prelude.String
```


In this example, we are attempting to apply an operator (+) that takes two arguments that must be of the same type to two arguments of different types (`Double` and `String`). Since the types of the arguments cannot be resolved with the type of the operator (more about this below), the type inferencer cannot determine a consistent type for the expression, and compilation fails.

The requirement that the type inferencer be able to unambiguously determine the type of an expression can sometimes cause unexpected behaviour:

```
5 * 8
Error: Ambiguous type signature in inferred type (Prelude.Num a,
Prelude.Outputable a) => a
```

This error message occurs because the type inferencer is not able to determine the exact type of `5` or `8`, and therefore the type of the whole expression. The best it can determine is that `5` and `8` are both members of some numeric type. However, `5` and `8` could each represent a value of several numeric types (including `Int`, `Integer`, and `Double`); there is no way to tell from the given form of the expression what specific type was intended:

```
:t 5 * 8
outputs Num a => a
```

There are a number of ways to deal with this behaviour. One is to declare the type of the entire expression; if the type inferencer is able to make type assignments that are consistent with this declaration, then the expression will compile.

The `::` operator is used to declare the type of expressions:

```
5 * 8 :: Integer
returns 40

5 * 8 :: Double
returns 40.0
```

Another approach is to declare the type of one or both of the subexpressions:

```
(5 :: Integer) * 8
returns 40
```

Since the first argument is an `Integer`, and the `*` operator accepts two arguments of the same type, the inferencer can determine that the other argument must also be an `Integer`, with the result being an `Integer` as well:

```
:t (5 :: Integer) * 8
outputs Integer
```

2.2.2 Lists

In addition to the simple types, CAL also includes several built-in types for combining values of the simple types. The two most important of these additional types are lists and tuples.

A list is an ordered collection of zero or more values of the same type. In CAL, lists are written surrounded by square brackets, with the elements separated by commas:

```
["list", "of", "strings"]
returns ["list", "of", "strings"]
```

```
["invalid", 'L', 10]
Error: Type error. All elements of a list must have compatible
types. Caused by: Type clash: type constructor Prelude.String
does not match Prelude.Char
```

List values can be constructed by adding elements to the front using the Cons operator (`:`). The Cons operator takes two arguments: An element to prepend to a list, and the list to prepend to, and returns a new list consisting of the element prepended to the provided list:

```
"a" : ["list", "of", "strings"]
returns ["a", "list", "of", "strings"]
```

```
5.0 : [4.0]
returns [5.0, 4.0]
```

List values can be combined using the append operator (`++`). The append operator takes two list arguments and returns a list consisting of the elements of the first list followed by the elements of the second list:

```
[1.0, 2.0, 3.0] ++ [0.4, 0.9]
returns [1.0, 2.0, 3.0, 0.4, 0.9]
```

It is possible for a list to have zero elements. The "empty list" is a valid list; it is represented by empty square brackets:

```
10.5 : []
returns [10.5]
```

```
['a', 'b', 'c'] ++ []
returns ['a', 'b', 'c']
```

```
[] ++ ["val1", "val2"]
returns ["val1", "val2"]
```

```
[]
Error: Ambiguous type signature in inferred type
Prelude.Outputable a => a
```

Note that the empty list produces an error when entered by itself. That is because the type inferencer cannot determine from the expression alone whether it refers to a list of `String`s, a list of `Double` values, or a list of some other type. As with ambiguous numeric types, an explicit type declaration can remove the ambiguity:

```
[] :: [Char]
returns []
```


The type of a list is represented by the type of its elements enclosed within square brackets:

```
:t ["Jack", "Tom"]
outputs [String]

:t ["Simone", "Sally", "Sarah"]
outputs [String]

:t [3.0, 1.0, 1.0, 2.0]
outputs [Double]

:t [[1.1, 1.2], [0.0, 0.3], [], [-3.0]]
outputs [[Double]]
```

The first two examples are both lists of strings. Notice that the two lists have the same type even though they have differing numbers of elements.

The third example is a list of `Double`s. The final example is a list of lists of `Double`s. Notice that it is possible for a list's elements to be lists themselves. Note also that a list of lists of `Double`s has a different type from a list of `Double`s, since their elements are of differing types.

In other words, you cannot mix lists of `Double`s with `Double`s in the same list:

```
[1.0, [1.2], 1.1]
Error: Type error. All elements of a list must have compatible
types. Caused by: Type clash: type constructor Prelude.Double
does not match Prelude.List
```

2.2.3 Tuples

Tuples are ordered collections of fixed numbers of values. The most familiar kind of tuple is the 2-tuple, or pair. Pairs or triples (3-tuples) are often used to model geometric locations:

```
(0.0, 0.0)

(-13.9, 50.0, 12.5)
```

Although the above examples are pairs and triples whose components are all of the same type (`Double`), it is also valid for tuple components to be of different types:

```
("James", 27.0)

('A', "Bob")
```

The type of a tuple is written as the types of each component within parentheses:

```
:t ("James", 27.0)
outputs (String, Double)

:t (61.0, "Bob")
outputs (Double, String)

:t (0.0, 0.0)
outputs (Double, Double)
```



```
:t (-13.9, 50.0, 12.5)
outputs (Double, Double, Double)
```

Note that the first tuple (consisting of a `String` followed by a `Double`) has a different type than the second tuple (consisting of a `Double` followed by a `String`). Note also that the third example (pair of `Double`s) has a different type than the fourth example (triple of `Double`s). This is because, unlike lists, every member of a given tuple type must have the same number of components.

Tuple components needn't be of simple types; they can be of any type, including other tuple types:

```
:t ("Origin", (0.0, 0.0))
outputs (String, (Double, Double))

:t (['a', 'b', 'c], 3.0)
outputs ([Char], Double)

:t ([('a', 1.0), ('b', 2.0), ('c', 3.0)], True)
outputs ([(Char, Double)], Boolean)
```

2.2.4 Case expressions, part 1

In addition to utility functions such as `fst`, `snd`, `head`, `tail`, and so forth that CAL provides for accessing the components of lists and tuples, CAL also provides a construct known as the case expression. Case expressions allow you to bind the components of lists and tuples to variable names to make accessing them more convenient:[1]

```
isOrigin :: Num a => (a, a, a) -> Boolean;
public isOrigin point =
    case point of
    (x, y, z) -> x==0 && y==0 && z==0;
    ;
```

In the above example, the case expression matches the `point` value. Each case expression contains one or more alternatives. An alternative is a pattern (for example, `(x, y, z)`) followed by a right-arrow (`->`) followed by an expression. The alternative whose pattern matches the case expression's argument is chosen, and the alternative's expression is evaluated as the value of the case expression.

The pattern `(x, y, z)` matches any 3-tuple. It binds the identifiers `x`, `y`, and `z` to the values of the first, second, and third components of the 3-tuple for the scope of the alternative's expression.

Similar patterns can be used for tuples of any number of dimensions. Elements that you don't wish to bind to an identifier can be filled in with an underscore (_):

[1] This is not all that case expressions are used for. See Case expressions, part 2 below.


```
myFst :: (a,b) -> a;
public myFst pair =
    case pair of
    (ret, _) -> ret;
    ;

mySnd :: (a,b) -> b;
public mySnd pair =
    case pair of
    (_, ret) -> ret;
    ;
```

Case expressions can also be used to match against list arguments:

```
any :: (a -> Boolean) -> [a] -> Boolean;
public any p list =
    case list of
    [] -> False;
    listHead : listTail -> p listHead || any p listTail;
    ;
```

The pattern `[]` matches the empty list. The pattern `listHead : listTail` matches any non-empty list. In the above example, the first element of the list is bound to `listHead`, and the rest of the list is bound to `listTail` in the second alternative.

2.3 Functions

A function is an operation that takes some number of values (possibly zero), and returns another, possibly different value. For example, the `max` function takes two arguments and returns the largest of the two:

```
max 1.5 0.3
returns 1.5
```

Note that function applications are a kind of expression. They must always produce a value, and their types must be consistent.

The `id` function takes a single value and returns an identical value:

```
id "ahoy"
returns "ahoy"
```

The `head` functions takes a single list value and returns the first element of the list:

```
head ["top", "middle", "bottom"]
returns "top"
```

The `pi` function takes no arguments and returns a `Double` value that approximates the value of pi:

```
pi
returns 3.141592653589793
```

The `isEven` function takes an `Int` argument and returns a `Boolean` indicating whether it is even (`True`) or odd (`False`):


```
isEven 9
returns False

isEven 10
returns True
```

The `power` function takes two `Double` arguments and raises the first argument to an exponent specified by the second argument, returning another `Double`:

```
power 2 0.5
returns 1.4142135623730951
```

The arguments to a function need not all be of the same type. For example, the `subscript` function in the `String` module accepts a `String` and an `Int` specifying an index into the `String`, and returns the `Char` at the specified index:

```
String.subscript "a modern Major-General" 9
returns 'M'
```

In CAL, functions themselves are values. Like all other values, they have a type:

```
:t pi
outputs Double

:t isEven
outputs Int -> Boolean

:t power
outputs Double -> Double -> Double

:t String.subscript
outputs String -> Int -> Char;

:t id
outputs a -> a

:t head
outputs [a] -> a
```

The first type signature is familiar; `pi` returns a value of type `Double`.

The second type signature contains a new element: It includes an arrow (->). The arrow indicates a function with arguments. The type signature `Int -> Boolean` specifies the type of a function that accepts a single `Int` argument and returns a `Boolean` value.

The type of functions that return multiple values are specified by listing the types of the arguments in order and separated by arrows, followed by another arrow and the type of the return value. So the type signature of `power` (`Double -> Double -> Double`) represents the type of a function that accepts two `Double` arguments and returns another `Double` value, and the type signature of `String.subscript` (`String -> Int -> Char`) represents the type of a function that accepts a `String` and an `Int` and returns a `Char`.

The fifth signature (for the `id` function) has another new element: It refers to an unfamiliar type `a`.

The type signature `a -> a` specifies the type of a function that accepts a value of type `a` and returns another value of the same type `a`, for some type `a`. The identifier `a` is a *type variable*; it can be bound to any type. Note that it can only be bound to a single type within a given type signature.

What this means is that we can pass an argument of any type at all to `id`, and it will return a value of the same type.

Similarly, the final type signature (for the `head` function) specifies a function that accepts a list of `a` (for some type `a`), and returns a value of the same type `a`.

It is possible to use multiple type variables in a single type signature. For example, the fst function accepts a 2-tuple, whose components can each be of any type, and returns the value of the first component. Similarly, the snd function accepts a 2-tuple with components of any type, and returns the second component:

```
fst ("foo", 2.0)
returns "foo"

snd ("foo", 2.0)
returns 2.0

:t fst
outputs (a, b) -> a

:t snd
outputs (a, b) -> b
```

Note that the type signature for `fst` (`(a, b) -> a`) contains two type variables. The first component of the argument can be of any type; the second component can also be of any type, and needn't be the same type as the first. But whatever the type of the first component is, the return value must be of the same type, because the first component and the return type have the same type variable (`a`).

Type variables are distinguished from specific type names by their case. Type variables always begin with a lower-case letter, whereas type names always begin with an upper-case letter. It is customary to choose type variable names starting with a and moving up the alphabet (`b`, `c`, etc.) as new names are required.

2.3.1 Type class constraints

Often, we don't want to restrict a value to a single type, but we don't want to allow it to be of just any type either. We want to restrict it to be a member of some group of types, such as the numeric types, or the types for which it is possible to compare for equality. For these situations, CAL provides a mechanism known as *type classes*.

A type class is a group of types that all implement some common set of operations (known as *methods*). There is generally an implicit semantic "contract" about the meaning of these operations. For example, in the `Eq` class, it is understood that if `equals x y` is `True`, then `notEquals x y` should be `False`, and vice versa, for all `x` and `y`.

We can specify that a type variable must belong to a given type class (or group of type classes). We do this by adding a type class constraint to a type signature. A type class constraint is represented by a type class name followed by the variable to which it applies, followed by a double arrow (`=>`), followed by the type signature that is being constrained:

```
:t max
outputs Ord a => a -> a -> a
```

The `max` function has a type that includes a type class constraint. Its type may be read as "a function that accepts two arguments of type `a` and returns a value of type `a`, for some type `a` that is a member of the `Ord` class".

2.3.2 Higher-order functions

We mentioned above that functions are themselves values. This means that it is possible for a function to accept another function as an argument, and/or to return functions. A function which operates upon other functions is called a higher-order function. Higher-order functions are a central technique of functional programming.

CAL provides a number of built-in higher-order functions in the standard library modules. Many, although not all, of these functions abstract various kinds of iterations over lists, arrays and other container types.

The `map` function accepts two arguments: A function that accepts a single argument, and a list of elements of the type that the function accepts. `map` returns a list of the results of applying the function argument to each element of the list argument:

```
:t map
outputs (a -> b) -> [a] -> [b]

map isEven [9, 8, 7, 6]
returns [False, True, False, True]

map head [['H', 'T'], ['1', '2'], ['a', 'b', 'c']]
returns ['H', '1', 'a']

map fst [("Q1",True), ("Q2",False), ("Q3",False), ("Q4", True)]
returns ["Q1", "Q2", "Q3", "Q4"]
```



```
map fst [['H', 'T'], ['1', '2'], ['a', 'b', 'c']]
Error: Type Error during an application. Caused by: Type clash:
type constructor Prelude.Tuple2 does not match Prelude.List.
```

The last call fails because the `fst` function accepts a tuple argument, but the list passed to `map` is a list of lists, not tuples.

The `filter` function accepts two arguments: a function that accepts a single argument and returns a `Boolean` value, and a list of elements of a type that the function accepts. It returns a list of each element for which the function argument returns `True`:

```
:t filter
outputs (a -> Boolean) -> [a] -> [a]

filter isEven [9, 8, 7, 6, 5, 4]
returns [8, 6, 4]

filter head [['H', 'T'], ['1', '2']]
Error: Type Error during an application. Caused by: Type clash:
type constructor Prelude.Boolean does not match Prelude.Char.
```

The final call fails, because `head` of a list of `Char`s returns a `Char` value, not a `Boolean` value.

The `compose` function accepts two single-argument functions and returns a new function equivalent to applying the first function to the result of the second function. The first function must accept as its argument values of the same type as the second function returns.

```
fst (1::Int, 'E')
returns 1

isEven 1
returns False

isEven (fst (1::Int, 'E'))
returns False

(compose isEven fst) (1::Int, 'E')
returns False

(compose fst isEven) (1::Int, 'E')
Error: Type Error during an application. Caused by: Type clash:
type constructor Prelude.Boolean does not match a record type.

:t fst
outputs (a, b) -> a

:t isEven
outputs Int -> Boolean

:t (compose isEven fst)
outputs (Int, a) -> Boolean
```


The attempt to call `(compose fst isEven)` fails, (even though the call to `(compose isEven fst)` succeeds) because `isEven` returns a `Boolean`, whereas `fst` accepts a 2-tuple as its argument.[2]

2.3.3 Defining functions

Functions are specified using a function definition. Function definitions are not expressions, so you cannot type them into the prompt in ICE. They must appear in a module, which may then be compiled and loaded.

Function definitions take the form of an equation, with an optional visibility specification:

```
public square n =
    n * n;

public removeEmptyElements list =
    filter (compose not isEmpty) list;
```

The first definition specifies that `square x` is a function that returns `x * x`. It is also public, which means that it can be used from outside the module in which it is defined. The second definition specifies that `removeEmptyElements x` is a function that accepts a list of lists and returns a new version of the list that has all of the empty lists removed.

With the `square` and `removeEmptyElements` functions defined as above, the following expressions can be evaluated:

```
square 2.1
returns 4.41

removeEmptyElements [['a', 'b', 'c'], [], ['d', 'e'], []]
returns [['a', 'b', 'c'], ['d', 'e']]
```

It is good practice to precede each function definition with a type declaration:

```
square :: Num a => a -> a;
public square x =
    x * x;

removeEmptyElements :: [[a]] -> [[a]];
public removeEmptyElements x =
    filter (compose not isEmpty) x;
```

In the first example, we declare that `square` accepts a single argument, which must be a member of the `Num` class, and returns a value of the same type. In the second example, we declare that `removeEmptyElements` accepts a single list of lists of some type `a`, and returns a list of lists of the same type `a`.

[2] `fst` has type `(a, b) -> a`, and `isEven` has type `Int -> Boolean`. It's possible to use the output from `fst` as the input to `isEven`, because we can treat the type variable `a` as referring to the `Int` type (in other words, the function constructed by compose will accept any 2-tuple whose first element is an `Int`). However, there is no way to use the output of `isEven` as input to `fst`, because there's no way to bind the type variables such that a `Boolean` value matches the type `(a, b)`.

2.3.4 Lambda expressions

It is often convenient to create functions that have no names. Such functions are referred to as lambda expressions. They are usually passed to higher-order functions such as `map` or `filter`. In CAL, lambda expressions are introduced by a backslash:

```
\x y -> (y, x)
```

The arguments of the function are listed after the backslash, and the body of the function is listed after the right-arrow (`->`). The above expression produces a function that accepts two arguments and returns a pair of the arguments in reversed order:

```
:t \x y -> (y,x)
outputs a -> b -> (b,a)
```

Anonymous functions are often used in situations where a very specific, single-purpose function is required:

```
maxList :: Ord a => [a] -> a;
maxList valueList =
    List.foldRight1 max valueList;

normalizeScores :: [Double] -> [Double];
normalizeScores rawScores =
    let
        maxScore :: Double;
        maxScore = maxList rawScores;
    in
        map (\x -> x / maxScore) rawScores;
```

In the example above, `normalizeScores` passes to `map` a lambda expression that accepts a single value and divides it by a specific number (in the case, the number returned by `maxList rawScores`). This function does not make recursive calls to itself, so it is both possible and convenient to pass it in as a lambda expression rather than creating a separate named function.

It is possible (and frequently desirable) to access local variables from the scope that an anonymous function is specified within. For example, in the above code, the lambda expression passed to `map` references the local definition of `maxScore`.

2.4 User-defined types

In addition to its various built-in types, CAL also allows you to define your own types using `data` definitions[3]. One such type defined in the Prelude is the `Ordering` type. We can write our own version:

[3] This is not the only way to define new types. See also Foreign type definitions below.


```
/**
 * Represents an ordering relationship between two values: less
 * than, equal to, or greater than.
 *
 * @see typeClass = Ord
 * @see function = compare
 */
data public MyOrdering =
    /**
     * A data constructor that represents the ordering
     * relationship of "less than".
     */
    public MyLT |
    /**
     * A data constructor that represents the ordering
     * relationship of "equal to".
     */
    public MyEQ |
    /**
     * A data constructor that represents the ordering
     * relationship of "greater than".
     */
    public MyGT
    deriving Eq, Ord, Enum, Bounded, Outputable;
```

This example defines the `MyOrdering` type as being `public` (i.e., usable in other modules). The `MyOrdering` type contains three values, which are constructed by the data constructors `MyLT`, `MyEQ`, and `MyGT`. Note that the data constructors are also declared as being `public`. This means that code in other modules will be able to use the data constructors directly to create values of this type (i.e., this is not an Abstract Data Type).

We could use the `MyOrdering` type to write functions that order geometric points:

```
comparePoints :: Num a => (a, a) -> (a, a) -> MyOrdering;
public comparePoints point1 point2 =
    case point1 of
    (x1,y1) ->
        case point2 of
        (x2,y2) ->
            if x1 > x2 then
                MyGT
            else if x1 < x2 then
                MyLT
            else if y1 > y2 then
                MyGT
            else if y1 < y2 then
                MyLT
            else
                MyEQ;
        ;
    ;
```

With `comparePoints` defined as above, the following expressions can be evaluated:


```
comparePoints (1.0, 2.0) (5.0, 6.0)
returns MyLT

comparePoints (1.0, 2.0) (-4.0, 1.0)
returns MyGT

comparePoints (1.0, 2.0) (1.0, 1.0)
returns MyGT

comparePoints (9.0, 9.5) (9.0, 9.5)
returns MyEQ
```

It is also possible to define types that "wrap" other types. Types of this kind have at least one data constructor that accepts one or more arguments:

```
data public MyMaybe a =
    public MyNothing |
    public MyJust
        value :: a;
```

The above defines a family of types `MyMaybe a` with two data constructors: `MyNothing`, which is a familiar 0-argument data constructor, and `MyJust`, which takes a single argument, named `value`, of type `a`.

The Prelude defines a type of this form called `Maybe a`. The return type from functions that might fail is often `Maybe Result` (where `Result` is some result type). If a database operation succeeds, for example, a function might return `Just` *`value`*, (where value is a value of type `Result`) whereas if it fails, it might return `Nothing`.

The `Maybe a` family contains types for all of the different bindings of `a`:

```
:t Just 50.0
outputs (Maybe Double)

:t Just (10 :: Int)
outputs (Maybe Int)

:t Just "ahoy"
outputs (Maybe String)

:t Nothing
outputs (Maybe a)

Just 50.0
returns Just 50.0

Nothing
Error: Ambiguous type signature in inferred type
Prelude.Outputable a => a
```

The attempt to evaluate the type of `Nothing` on its own fails, because that data constructor does not take enough arguments to allow the type inferencer to determine the type represented by `a`. As usual, an explicit type declaration can resolve the ambiguity:

```
(Nothing :: Maybe Char)
returns Nothing
```


Note that we must declare the name and type of every argument that a data constructor takes. Note also that when the type of a data constructor's argument is a type variable, then that type variable must also appear as an argument to the type constructor:

```
data public MyEither a b =
    public MyLeft     value :: a |
    public MyRight    value :: b;
```
compiles without error

```
data public Value =
    public StringValue    strValue :: String |
    public IntValue       intValue :: Int |
    public BooleanValue   boolValue :: Boolean |
    public DoubleValue    dblValue :: Double;
```
compiles without error

The declaration of `Value` compiles without error even though its data constructors accept arguments that are not arguments to the type constructor because none of the argument types are type variables.

```
data public Broken1 =
    public Simple |
    public BrokenWildcard    arg :: a;
```
Error: The type variable a must appear on the left-hand side of the data declaration.

The declaration of `Broken1` fails because the `BrokenWildcard` data constructor takes an argument of type `a`, but the type constructor does not accept `a` as an argument.

```
data public Broken2 a =
    public BrokenLeft     value :: a |
    public BrokenRight    value :: b;
```
Error: The type variable b must appear on the left-hand side of the data declaration.

The declaration of `Broken2` fails because the data constructor `BrokenRight` takes an argument of type `b`, but the type constructor does not take `b` as an argument.

2.4.1 Case expressions, part 2

As with lists and tuples, the components of user-defined data structures are accessed using case expressions. Patterns based on the data constructors for a type can be used to unpackage the components of an instance of a user-defined type:

```
maybeToList :: Maybe a -> [a];
public maybeToList m =
    case m of
    Nothing -> [];
    Just value  -> [value];
    ;
```

The above code converts any value of type `Maybe a` (for some type `a`) into a list of elements of type `a`. If the value passed in is `Nothing`, then the list is empty. If the value passed in is `Just` *x* (for some value *x*), then the list will be a single-element list whose only element is *x*. With the above definition, we can evaluate the following expressions:

```
maybeToList (Just 'c')
returns ['c']

maybeToList (Nothing :: Maybe Int)
returns []

maybeToList (Just 12.2)
returns [12.2]
```

If only certain components in a user-defined type are required, an alternative syntax may be used specifying only the required components:

```
data public TripleType a b c =
    public TripleDC
        field1 :: a
        field2 :: b
        field3 :: c;

public addFirstAndThird tripleType =
    case tripleType of
    TripleDC {field1, field3} -> field1 + field3;
    ;
```

The `addFirstAndThird` function returns the sum of the values of the `field1` and `field3` arguments from a `TripleType` data value.

```
addFirstAndThird (TripleDC 2.0 "String" 3.0)
returns 5.0
```

There are times when we don't want to specify all possible alternatives in a case expression. In these instances, we can use a default pattern (the underscore) to match all of the alternatives that are not explicitly specified:

```
pointGreaterThan :: Num a => (a, a) -> (a, a) -> Boolean;
pointGreaterThan p1 p2 =
    case comparePoints p1 p2 of
    MyGT -> True;
    _ -> False;
    ;
```

For the purposes of `pointGreaterThan`, all that matters is whether `comparePoints` returns `MyGT`. If it does not, then it doesn't matter whether it returned `MyLT` or `MyEQ`. So rather than specify separate patterns for `MyLT` and `MyEQ`, the above definition provides only two patterns: One for `MyGT`, and one for everything else.

2.5 Accessing CALDoc information

CAL has a special kind of comment known as CALDoc that is similar to Javadoc. CALDoc comments are added by the developer to CAL code elements, which are processed by the compiler during compilation.

The source in the CAL standard library has a lot of CALDoc associated with it. This CALDoc documentation can be a very useful source of up-to-date documentation on specific functions, data constructors, methods, types, type classes, and modules.

ICE provides commands for displaying the CALDoc associated with CAL entities:

ICE Command	Description
`:docm` *module_name*	Show the CALDoc comment associated with the named
`:docf` *function_or_class_method_name*	Show the CALDoc comment associated with the named function or class method.
`:doct` *type_name*	Show the CALDoc comment associated with the named type constructor
`:docd` *data_constructor_name*	Show the CALDoc comment associated with the named data constructor.
`:docc` *type_class_name*	Show the CALDoc comment associated with the named type class.
`:doci` *type_class_name instance_type_constructor*	Show the CALDoc comment associated with the named instance.
`:docim` *method_name type_class_name instance_type_constructor*	Show the CALDoc comment associated with the named instance method.

For example, to display the CALDoc associated with the Dynamic module, use the `:docm` command from the ICE command prompt:

```
:docm Dynamic
outputs
CALDoc for the Cal.Core.Dynamic module:

Defines the Dynamic type along with a variety of functions for
working with it.

Based on the Dynamics module in Hugs and GHC, and the paper
"Scrap your boilerplate: a practical design for generic
programming".

Author:
  Bo Ilic
  James Wright
```

Similarly, to display the CALDoc associated with the `minBound` method, use the `:docf` command:


```
:docf minBound
outputs
Cal.Core.Prelude.minBound :: Bounded a => a

CALDoc for the Cal.Core.Prelude.minBound class method:

(Required method)

Returns:
  result :: Bounded a => a
    the minimum bound of the instance type.
```

Note that it is possible for the methods of a specific instance to have their own CALDoc also. To view this CALDoc, use the `:docim` command. For example, to see the CALDoc for the `minBound` method of the `Bounded` instance for the `Char` type:

```
:docim minBound Bounded Char
outputs
minBound :: Char

CALDoc for the minBound method in the
Cal.Core.Prelude.Bounded#Cal.Core.Prelude.Char instance:

(Required method)

The minimum bound for Char is '\u0000'.

Returns:
  result :: Char
```

2.6 Standard library modules

The CAL standard library is divided into several different modules. Each module provides functions, types, type classes and/or instances related to a specific area of functionality.

Every CAL module must import the Prelude module. If you wish to use functionality provided by any other library module, then you must import that module as well. For example, to operate on `Array` values a CAL module must import at least the Prelude module and the Array module.

2.6.1 Cal.Core.Prelude

The Prelude module is the core module of CAL. It must be imported by every other CAL module.

The Prelude defines the primitive types `Char`, `Boolean`, `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double` that correspond to the primitive unboxed Java types. It also defines other important CAL types such as `String`, `Function`, `List`, `Maybe`, `Either`, `Unit`, and the built-in record and tuple types.

The Prelude also defines the core type classes: `Eq`, `Ord`, `Num`, `Inputable`, `Outputable`, `Appendable`, `Bounded`, `Enum`, `IntEnum`, and `Typeable`, as well as appropriate instances of these classes for the types listed above.

The Prelude also contains definitions for many functions that are widely useful in writing CAL code.

2.6.2 Cal.Collections.Array

The Array module defines the `Array` abstract data type, plus a variety of functions and instances for operating on `Array` values.

`Array` is a polymorphic type; you can construct an `Array` of CAL values of any type. `Array` values are immutable (ie, purely functional); all operations that "change" element values actually return an entirely new `Array` value without modifying the original.

`Array` values allow constant time access to their elements.

2.6.3 Cal.Core.Bits

The Bits module defines the `Bits` type class. The `Bits` type class has methods for performing bitwise operations on numeric values (eg, shifts, bitwise boolean operations, etc.). It also provides `Bits` instances for the `Int` and `Long` types.

2.6.4 Cal.Core.Char

The Char module defines a number of useful library functions for operating on `Char` values (eg `isLetter`, `isLowerCase`).

2.6.5 Cal.Core.Debug

The Debug module provides functionality for debugging CAL programs. It provides the `Show` type class, whose `show` method that will convert a CAL value to a human-readable string value, as well as `Show` instances for the main CAL data types.

It also provides the `trace` function, which is useful for inserting tracing statements into CAL code (for "`printf` debugging").

The Debug module also provides functions to support execution timing and reduction tracing.

2.6.6 Cal.Utilities.Decimal

The Decimal module provides useful libraries functions for the `Decimal` type. The `Decimal` type is an arbitrary-precision integer type based on the Java `BigDecimal` type.

2.6.7 Cal.Core.Dynamic

The Dynamic module defines the `Dynamic` type and functions for operating on `Dynamic` values. `Dynamic` values can contain CAL values of any type; this allows for the writing of dynamically-typed code (for example, lists that contain elements of more than one type). See Dynamic typing on page 104 for more details.

2.6.8 Cal.Core.Exception

The Exception module provides support for handling exceptions in CAL.

Exceptions can arise in CAL in a number of ways:

1. a Java exception is thrown by a call to a foreign function or a primitive function
2. a call to the error function
3. a pattern matching failure in a case expression or data constructor field selection expression e.g. `(case Just 10.0 of Nothing -> "abc";)` and `(Left 10.0).Right.value`.
4. a call to the `throw` function

Exceptions in categories 1-3 are called Java exceptions because they are not associated with a specific CAL type. They can be caught with handlers for the type `Exception.JThrowable`. They can also be caught with any CAL foreign type that is an instance of the type class `Exception` such that the foreign implementation type is a Java subclass of java.lang.Throwable, and such that the exception is assignment compatible with the implementation type. For example, if a foreign function throws a java.lang.NullPointerException, then this can be caught by a handler for the CAL types `JThowable`, `JRuntimeException`, `JNullPointerException`, but not a handler for `JIllegalStateException` (assuming the natural implied data declarations for these CAL types).

Exceptions in category 4 are called CAL exceptions because they are associated with a specific CAL type, namely the type at which the exception was thrown using the throw function. They must be caught at that precise type.

Here is an example showing a function throwing a CAL value of a record-type as an exception, catching it, and then doing some simple manipulations:


```
calThrownException5 =
    throw ("abc", 1 :: Int, 2 :: Integer,
           ["abc", "def"], Just (20 :: Int))
    `catch`
    (
        let
            handler :: (String, Int, Integer,
                        [String], Maybe Int) -> String;
            handler r = show (r.#5, r.#4, r.#3, r.#2, r.#1);
        in
            handler
    );

//evaluates to True
testCalThrownException5 =
    calThrownException5 ==
        "(Prelude.Just 20, [\"abc\", \"def\"], 2, 1, \"abc\")";

instance Exception String where;

instance Exception Int where;

instance Exception Integer where;

instance Exception a => Exception (Maybe a) where;

instance Exception a => Exception [a] where;

instance Exception r => Exception {r} where;
```

2.6.9 Cal.Collections.Map, Cal.Collections.IntMap, Cal.Collections.LongMap

The Map module defines the `Map` type, which is an efficient implementation of maps from keys of arbitrary type to values of arbitrary type. The `IntMap` and `LongMap` modules provide the `IntMap` and `LongMap` types, which are efficient implementations of maps from `Int` or `Long` keys (respectively) to values of arbitrary type.

2.6.10 Cal.Collections.List

The List module provides useful library functions for operating on List values. The actual List type is defined in the Prelude module.

2.6.11 Cal.Utilities.Locale

The Locale module defines the `Locale` type, and provides functions for working with locale values, accessing locale properties of the system, and performing locale-sensitive string comparisons through the use of the types `Collator` and `CollationKey`.

2.6.12 Cal.Utilties.Math

The Math module provides a number of useful library functions for math, such as trigonometric, logarithmic, and exponential operations, among others.

2.6.13 Cal.Utilities.MessageFormat

The MessageFormat module defines a set of functions for formatting strings with message patterns. It provides a means for producing concatenated messages in a localizable way.

2.6.14 Cal.Utilities.QuickCheck

The QuickCheck module provides a simple mechanism for testing programs by generating arbitrary test data.

The basic idea is to help simplify and improve testing by automatically creating arbitrary test data. Properties of programs can be described by functions. QuickCheck can be used to validate these properties by automatically generating lots of arbitrary data to fill the functions' parameters. If a property is not true, QuickCheck can report the parameters that falsified the property.

2.6.15 Cal.Utilities.Random

The Random module provides functions for generating lists of pseudo-random numbers.

2.6.16 Cal.Core.Record

The Record module provides many useful functions for working with CAL record types. Since tuples are records, these functions are also useful for working with tuples.

2.6.17 Cal.Core.Resource

The Resource module provides access to localizable user resources in the CAL environment.

2.6.18 Cal.Collections.Set

The Set module provides the Set type, which is an efficient implementation of sets of values.

2.6.19 Cal.Core.String

The String module provides a number of useful library functions for operating on String values.

2.6.20 Cal.Utilities.StringNoCase

The StringNoCase module defines the `StringNoCase` type, which represents case-insensitive string values, as well as appropriate type class instances for the `StringNoCase` type. It also provides functions for operating on `StringNoCase` values and for converting between `String` and `StringNoCase` values.

There aren't a lot of functions for operating on `StringNoCase` values, because converting between `String` and `StringNoCase` values is quite efficient. That means that it's usually feasible to just convert to a `String`, use the appropriate `String` operation, and then convert back to a `StringNoCase`.

2.6.21 Cal.Utilities.StringProperties

The StringProperties module defines the types `StringProperties` and `StringResourceBundle` with are useful for working with string resource files.

2.6.22 Cal.Core.System

The System module provides functions for interacting with the current CAL execution context. Among the functions it provides are `getProperty`, `hasProperty`, and `propertyKeys`, which can be used to query the execution context for CAL system properties.

2.6.23 Cal.Utilities.TimeZone

The TimeZone module defines the `TimeZone` type and its affiliated operations.

3 Language Reference

3.1 Comments

Comments in CAL can be classified into two categories: regular comments and CALDoc comments.

3.1.1 Regular comments

A regular comment is a piece of human-readable text that is ignored by the compiler. CAL uses the same syntax for regular comments as Java and C++. The compiler will ignore any text that falls between two forward-slashes (`//`) and the end of the line, or that falls between a forward-slash and an asterisk (`/*`) and an asterisk and a forward-slash (`*/`).

Ex:

```
5.0         // Compiler ignores this text, but processes the 5.0

/* Comments in the slash-star style
   may span
   multiple lines */
```

3.1.2 CALDoc comments

A CALDoc comment is a piece of end-user and developer visible documentation in the source code. Similar to a Javadoc comment in Java, a CALDoc comment is delimited by a forward-slash and two asterisks (`/**`) and an asterisk and a forward-slash (`*/`). On each of the lines making up a CALDoc comment, any leading whitespace and subsequent asterisks (`*`) are ignored.

For more information on CALDoc comments, please see the section entitled CALDoc on page 80 in this document.

3.2 Expressions

An expression is a unit of code that can be reduced to a value. Every expression returns a value of a specific type. This section lists the various types of legal CAL expressions, with some examples.

3.2.1 The unit value

The simplest expression in CAL is the unit value. It corresponds to the `void` type of Java (and other languages). It is written as an empty pair of round brackets:

```
()
```


3.2.2 Numeric literals

Numeric literals can be written either in exponential notation, or in standard notation. Standard notation is written as a series of digits, with an optional decimal point:

```
digits[.[digits]]
```

Some examples:

```
50
50.0
```

Exponential notation is also permitted:

```
digits[.[digits]] e digits[.[digits]]
```

In exponential notation, a number is represented by one number (called the mantissa) multiplied by some power of ten. In CAL as in most languages, this is written as the mantissa followed by a letter E followed by the power of ten to multiply by:

```
5.0e2       // 5.0 * 10^2    == 500.0
5.0E-1      // 5.0 * 10^(-1) == 0.5
```

Note that the case of the letter E is not significant.

3.2.3 List expressions

A list is an ordered, variable-size series of values of the same type. In CAL, lists are represented by comma-separated values enclosed within square brackets:

Value	**Description**
`[]`	Empty list Type: `[a]`
`["foo", "bar", "baz"]`	List of three Strings Type: `[String]`
`[[1, 2], [9, 9, 8]]`	List of two lists (see Tuples) Type: `Num a => [[a]]`
`[(6.0, 7.0), (2.0, 1.0)]`	List of two pairs Type: `[(Double, Double)]`

Note that it is possible to have an empty list (i.e. a list containing 0 elements). It is also possible to nest lists, where one list contains other lists.

See Lists on page 90 for some of the specialized list-handling operations that CAL provides.

3.2.4 Record expressions

A record is an unordered collection of named values of possibly differing type. A record is represented by a comma-separated list of field assignments:

```
{fieldname = expression [, fieldname = expression ...] }
```

Ex:

Value	Description
`{}`	Empty record Type: `{}`
`{x=1.0, y=1.0}`	Record with two fields Type: `{x :: Double, y :: Double}`
`{name="Jack", age=32.0, gender='M'}`	Record with three fields Type: `{age :: Double, gender :: Char, name :: String}`
`{age=32.0, gender='M', name="Jack"}`	Equivalent to previous record.

Note that it is possible for a record to contain 0 fields (i.e., to be empty).

A fieldname is either an identifier starting with a lower-case letter, or an ordinal number preceded by the number sign (`#`).

It is possible to obtain the value of an individual field of a record expression using the field-selection operator (`.`). Attempting to obtain a field from a record that doesn't include the specified field is an error. Ex:

```
{ x=1.0, y=0.5 }.y
returns 0.5

{ name="Fred", age=22.0, #1='a', #2='b' }.#1
returns 'a'

{ name="Jack" }.age
Error: Type error. Invalid record selection for field age. Caused
by: the record type {name :: Prelude.String} is missing the
fields [age] from the record type a\age => {a | age :: b}.
```

It is also possible to extend a record with additional fields, and to update existing fields in a record. Both record extension and record update can be achieved by using a record case expression, or by using a special syntax.

Record extension:

```
{{x = 1.0, y = 1.0} | z = 2.0}
returns {x = 1.0, y = 1.0, z = 2.0}
```



```
case {x = 1.0, y = 1.0} of {x = x1, y = y1} -> {x = x1, y = y1, z
= 2.0};
returns {x = 1.0, y = 1.0, z = 2.0}
```

Record update:

```
{{x = 1.0, y = 1.0} | y := 2.0}
returns {x = 1.0, y = 2.0}

case {x = 1.0, y = 1.0} of {x = x1, y = _} -> {x = x1, y = 2.0};
returns {x = 1.0, y = 2.0}
```

3.2.5 Tuple expressions

A tuple is an ordered, constant-size collection of values of possibly differing type. Tuples are represented in CAL by comma-separated values enclosed within parentheses:

Value	Description
`(1.0, 2.0)`	Pair (2-tuple) of Doubles Type: `(Double, Double)`
`('a', 'b', 3.0)`	Triple (3-tuple) of two Chars and a Double Type: `(Char, Char, Double)`
`("s", [], [7.0], 'c')`	4-tuple of a String, two lists, and a Char Type: `(String, [a], [Double], Char)`

A tuple is a special case of a record whose fields have ordinal names from `#1` to `#n` (for a tuple of dimension *n*). So for example, a 3-tuple is equivalent to a record of three fields named `#1`, `#2`, and `#3`.

Since tuples are just a special case of records, all of the same syntax that applies to records can also be used with tuples. Ex:

```
{ #1=10.0, #2='a' }.#1
returns 10.0

(10.0, 'a').#1
returns 10.0

("s", [], "s2").#3
returns "s2"

("s", [], "s2").#4
Error: Type error. Invalid record selection for field #4. Caused
by: the record type (Prelude.Char, Prelude.Char, Prelude.Char) is
missing the fields [#4] from the record type a\#4 => {a | #4 ::
b}.
```


3.2.6 Character literals

CAL characters are represented by a character within single quotes:

`'A'`	Capital A
`'0'`	Digit 0
`'\t'`	Tab character
`'\''`	Single-quote

As with strings (see below), all of the usual escape-sequences work.

3.2.7 String literals

A string is an ordered sequence of characters. CAL string literals are written enclosed by double-quotes:

```
"this is a string"
"This \"string\" prints\n on two lines"
```

All of the usual escape-sequences work:

```
\n          newline
\r          carriage return
\t          tab
\"          double quote
\'          single quote
\\          backslash
\045        the character specified by
            the octal value 45 (i.e.
            '%')
\u0025      the character specified by
            the hexadecimal value 25
            (i.e. '%')
```

3.2.8 Function application

Functions are applied in CAL by writing the function name followed by the arguments separated by spaces:

```
fcn_name [arg1 [arg2 ... [argn]...]]
```

There is no special operator that represents function application. In particular, the list of arguments is not surrounded by parentheses. Example:

```
pi
returns 3.141592653589793

sin 0.0
returns 0.0

append "first string" " second string"
returns "first string second string"
```


Function application has higher precedence than any operator, so parentheses may occasionally be required to make a function call behave as expected. For example, the (somewhat contrived) call

```
let
    x = 9.0;
in
    max x + 5.0 6.0
Error: Type error applying the operator "+" to its first
argument. Caused by: Type clash: type Prelude.Double ->
Prelude.Double is not a member of type class Prelude.Num
```

will not take the maximum of `x + 5.0` and `6.0`. In fact, it will produce an error, because it parses as `(max x) + (5.0 6.0)` rather than (`max (x + 5.0) 6.0`). Parentheses around the first argument can correct this:

```
let
    x = 9.0;
in
    max (x + 5.0) 6.0
returns 14.0
```

Partial function application

One of the advantages of (and motivations for) CAL's syntax for function application is the ease with which it allows for partial function application. Partial function application allows one to create a new function with a smaller argument list from an existing function. For example, we can make a new version of `max` that already has one argument specified:

```
newMax = max 10.0;
```

This produces a new function of one argument that returns 10.0 for numbers less than 10.0, and unchanged numbers for numbers >= 10.0:

```
newMax 6.0
returns 10.0

newMax (-2.0)
returns 10.0

newMax 12.0
returns 12.0
```

3.2.9 Operator application

An operator is a function that is applied using infix notation. In other words, the name of the function is written in between its arguments instead of in front of them. For example, in the following expression the operator is `+` (addition):

```
1.0 + 2.0
returns 3.0
```

In this example, the operator is `*` (multiplication):

```
4.0 * 5.0
returns 20.0
```


CAL has the following operators:

Operator	Meaning	Equivalent function, data constructor, or class method
==	Equal to	Prelude.equals
!=	Not equal to	Prelude.notEquals
>	Greater than	Prelude.greaterThan
<	Less than	Prelude.lessThan
>=	Greater than or equal to	Prelude.greaterThanEquals
<=	Less than or equal to	Prelude.lessThanEquals
+	Addition	Prelude.add
-	Subtraction	Prelude.subtract
*	Multiplication	Prelude.multiply
/	Division	Prelude.divide
%	Remainder	Prelude.remainder
&&	Boolean AND	Prelude.and
||	Boolean OR	Prelude.or
++	Append	Prelude.append
:	List construction	Prelude.Cons
#	Function composition	Prelude.compose
$	Function application	Prelude.apply

Precedence and Associativity

When multiple infix operators are used in an expression, ambiguity can arise about the order in which they should be applied. For example, the expression

```
1.0 + 2.0 * 2.0
```

is ambiguous in the absence of a well-defined operator precedence; it can be interpreted as either (1.0 + 2.0) * 2.0, in which case its value is 6.0, or as 1.0 + (2.0 * 2.0), in which case its value is 5.0. Fortunately, there is a well-defined operator precedence that determines the order in which operations are performed. Operators with higher precedence are applied first. In CAL, as in standard arithmetic, the value of the above expression is 5.0, since the multiplication operator (*) has a higher precedence than the addition operator (+).

Arithmetic operators in CAL have the standard arithmetic precedence. There are also a number of other infix operators. The complete list of operator precedence is below.

Some elements of CAL syntax (:: for type expressions, backquotes (``) for arbitrary infix functions, and . for record and data constructor field selection) are "operator-like", in that they have associativity and precedence as well. The precedence table below includes these operator-like entities.

Operator groups higher in the list have higher precedence; operators in the same row have the same precedence:

Operator (or operator-like entity)	Description	Associativity
.	Record or data constructor field selection	Left-associative
`function` #	Backquoted-function operator, function composition	Left-associative (`function`), Right-associative (#)
-	Unary negation	Non-associative
* / %	Multiplication, division, remainder	Left-associative
+ -	Addition, subtraction	Left-associative
: ++	List construction, list appending	Right-associative
< <= == != >= >	Comparison operators	Non-associative
&&	Boolean AND	Right-associative
||	Boolean OR	Right-associative
$	Function application	Right-associative
::	Type signature	Non-associative

Operators that have the same precedence are grouped in an expression on the basis of their associativity. Associativity in this context refers to whether the operators on the right are evaluated first, or those on the left. For example, consider the case of multiple divisions:

```
30.0 / 20.0 / 10.0
```

Without using associativity, this expression is ambiguous; it can be interpreted as either 30 / (20.0 / 10.0), in which case its value is 15.0, or as (30.0 / 20.0) / 10.0, in which case its value is 0.15. The first case is a right-associative grouping, whereas the second is a left-associative grouping. In CAL, the division operator is left-associative, so the expression above will evaluate to 0.15.

The precedence table above includes the associativity of each group of operators.

Note that the backquoted-function operator and the function composition operator are at the same precedence level, but they differ in associativity. This means that they cannot be used together in a single infix expression. For example, expressions of the forms

```
a `b` c # d
```

and

```
a # b `c` d
```

are syntactically invalid in CAL.

Operators for function composition and function application (# and $)

In CAL, the function composition operator (`#`) can be used to compose two functions using an infix operator expression. For example, the expression:

```
List.filter (not # isEmpty) listOfLists
```

returns a list of the non-empty elements in the list `listOfLists` by using the predicate `(not # isEmpty)`, a composed function. This operator is right-associative, and so the expression:

```
(f # g # h) x
```

is equivalent to:

```
(f # (g # h)) x
```

and is thus also equivalent to:

```
f (g (h x))
```

While the function application operator (`$`) is defined simply as:

```
f $ x = f x
```

its distinguishing feature is that it has a low precedence – lower than any other operator except the type signature operator (`::`). It can thus be used to construct nested expressions without parentheses. For example, while the expression:

```
f x y z
```

applies the function `f` to the arguments `x`, `y` and `z`, the expression:

```
f x $ y z
```

is equivalent semantically to the expression:

```
f x (y z)
```

which applies the function `f` to the arguments `x` and `(y z)`.

The function application operator is often employed idiomatically to chain a sequence of function applications together. For example, rather than writing:

```
toJIterator (map fromJust (filter isJust (reverse list)))
```

one can write:

```
toJIterator $ map fromJust $ filter isJust $ reverse list
```

Incidentally, the above is semantically equivalent to:

```
(toJIterator # map fromJust # filter isJust) (reverse list)
```

and is thus also equivalent to:

```
toJIterator # map fromJust # filter isJust $ reverse list
```


Backquoted functions

The usual way to apply functions is in prefix order:

```
subscript (subscript listOfLists 3) 5
```

However, in some circumstances (such as when chaining several calls together), it can be more clear to write a function application in infix order. CAL allows you to apply any function of two or more arguments using infix notation by surrounding the function name in backquotes (`). For example, the following is equivalent to the expression above:

```
listOfLists `subscript` 3 `subscript` 5
```

Note that the backquoted function must accept no fewer than two arguments. However, it may accept more. In the case where it accepts more than two arguments, the result is a partial application:

```
:t ['a'] `zip3` ['b']
outputs [a] -> [(Char, Char, a)]

(['a'] `zip3` ['b']) ['c']
returns [('a', 'b', 'c')]

1.0 `negate` 2.0
Error: Type Error during an application. Caused by: Type clash:
type constructor Prelude.Double does not match Prelude.Function.
```

The third expression fails because `negate` accepts only one argument.

3.2.10 If expressions

An if expression takes one of two different values depending on the boolean value of a test expression. It has the form

```
if condition_expression then
    if_true_expression
else
    if_false_expression
```

If *condition_expression* evaluates to `True`, then the value of the if expression is *if_true_expression*. Otherwise, it is the value of *if_false_expression*. Note that the `else` clause is not optional.

Ex:

```
if x > 0 then
    "positive"
else
    "negative or zero"
```

3.2.11 Case expressions

Case expressions provide a means to select from a variety of alternative expressions depending upon the value of a test expression. They are often used for extracting components of compound data structures such as fields, lists, tuples, or members of algebraic types.

A case expression consists of a specification of the expression to test followed by a list of alternatives. An alternative consists of a pattern to match followed by a right arrow (->) followed by an expression whose body will provide the value of the case expression if the alternative is selected. Variables in the pattern are bound in the corresponding expression.

```
case condition_expression of
pattern -> expression1 ;
[pattern -> expression2 ; ... ]
```

Ex:

```
myListSubscript :: [a] -> Int -> a;
public myListSubscript !list !index =
    case list of
    x : xs ->
        if index == 0 then
            x
        else if index > 0 then
            myListSubscript xs (index - 1)
        else
            error "negative index.";
    [] ->
        error "index out of bounds.";
    ;
```

With `myListSubscript` defined as above:

```
myListSubscript ['a', 'q', 'b'] 0
returns 'a'

myListSubscript ['a', 'q', 'b'] 2
returns 'b'

myListSubscript ['a', 'q', 'b'] 3
Error: index out of bounds.
```

Often, one doesn't care about the value of each component that a pattern matches. In these situations, it is possible to use the underscore character as a wildcard identifier. The following two examples are equivalent, except that the second does not make an unnecessary binding of the first list element:

```
length1 :: [a] -> Int;
public length1 !list =
    let
        lengthHelper !list !acc =
            case list of
            first : rest -> lengthHelper rest (acc + 1);
            [] -> acc;
            ;
    in
        lengthHelper list 0;
```

```
length2 :: [a] -> Int;
public length2 !list =
    let
        lengthHelper !list !acc =
            case list of
            _ : rest -> lengthHelper rest (acc + 1);
            [] -> acc;
            ;
    in
        lengthHelper list 0;
```

So with `length1` and `length2` defined as above:

```
length1 ['x', 'y', 'z', 'a', 'b']
returns 5

length2 ['x', 'y', 'z', 'a', 'b']
returns 5
```

Matching lists

There are two forms of pattern for matching lists:

`[]`	Matches an empty list
`headPattern : tailPattern`	Matches a non-empty list

The first form matches empty lists. The second form matches any non-empty list. It binds the first pattern to the head of the list (i.e., the first element), and the second pattern to the tail of the list (i.e., a list containing all of the elements of the original list except the first).

```
extractTail :: [a] -> [a];
public extractTail list =
    case list of
    [] -> error "empty lists have no tail";
    x : y -> y;
    ;
```

If `extractTail` is defined as above, then:

```
extractTail []
Error: empty lists have no tail

extractTail ['a']
returns []

extractTail ['a', 'b']
returns ['b']

extractTail ['a', 'b', 'c']
returns ['b', 'c']
```


Matching tuples

There is one specialized form of pattern for matching tuples.[4] To match a pair, enclose two patterns separated by a comma in parentheses. To match an *n*-tuple, enclose *n* patterns separated by commas in parentheses:

`(a, b)`	Matches a pair
`(a, b, c)`	Matches a triple
`(x, y, z, w)`	Matches a 4-tuple

Ex:

```
public vectorLength vector =
    case vector of
    (x, y) -> Math.sqrt (x * x + y * y);
    ;
```

With `vectorLength` defined as above,

```
vectorLength (3.0, 4.0)
returns 5.0

vectorLength 3.0 4.0
Error: Type Error during an application. Caused by: Type clash:
type constructor Prelude.Double does not match Prelude.Function

vectorLength (5, 12)
returns 13.0
```

The second expression raises an error, because `vectorLength` accepts a single 2-tuple, not two non-tuple values.

Interestingly, the third expression does not raise an error even though we use an ambiguous form of numeric literal. This is because the `sqrt` function has the type `Double -> Double`, so the type inferencer knows that its argument must be a `Double`. From this information, it can work back to deduce that the two components of the 2-tuple argument to `vectorLength` must also be `Double`s.

Note that a case expression that matches tuples can contain only a single alternative.

Matching records

There are three forms of pattern for matching records:

```
{}

{field_name1 [= pattern1] [, field_name2 [= pattern2] ...]}

{ pattern0 | field_name1 [= pattern1] [, field_name2 [= pattern2]
...]}
```

[4] Since tuples are just a special case of records, it is also possible to use the record-matching syntax to match tuples. See Matching records below.

The first form matches empty records only. The following function will signal an error if it is called with anything other than an empty record:

```
emptyRecordOnly :: {} -> String;
emptyRecordOnly record =
    case record of
    {} -> "empty record";
    ;
```

The second form matches a record that contains exactly the specified fields. The following function will signal an error if it is called with anything other than a record that contains a name field, an age field, and nothing else:

```
showNameAgeRecord :: {name :: String, age :: Double} -> String;
showNameAgeRecord record =
    case record of
    {name = nameValue, age = ageValue} ->
        concat ["Name =", nameValue, ". Age =", doubleToString
ageValue];
    ;
```

In the above example, the values of the `name` and `age` fields are bound to `nameValue` and `ageValue` respectively. However, it is not necessary to specify a new name for fields. If the new name is omitted, then the field will be bound to its own name. For example, the following definition of showNameAgeRecord is equivalent to the above definition:

```
showNameAgeRecord :: {name :: String, age :: Double} -> String;
showNameAgeRecord record =
    case record of
    {name, age} ->
        concat ["Name =", name, ". Age =", doubleToString age];
    ;
```

The values of the fields are bound to their own names in this version. This technique is known as "punning". Punning is forbidden for ordinal fields. You must always specify a new name for ordinal fields of a record if you wish to unpackage them using a case expression.

In the following example, we bind the fields of the first record to `name1` and `age1` and the fields of the second record to `name2` and `age2`. This allows us to access the fields of both records at the same time. If we had used punning (i.e., had not provided new names for the fields), then the binding for the `name` field of `record2` would have masked the binding for the `name` field of `record1`.


```
equalsNameAgeRecord :: {name :: String, age :: Double} -> {name
:: String, age :: Double} -> Boolean;
equalsNameAgeRecord record1 record2 =
    case record1 of
    {name = name1, age = age1} ->
        case record2 of
        {name = name2, age = age2} ->
            name1 == name2 && age1 == age2;
        ;
    ;
```

With `equalsNameAgeRecord` defined as above:

```
equalsNameAgeRecord {name = "Matt", age = 54} {age = 54, name =
"Matt"}
returns True
```

```
equalsNameAgeRecord {name = "Jack", age = 54.0, profession =
"blacksmith"} {name = "Jack", age = 54.0, profession =
"blacksmith"}
Error: Type Error during an application. Caused by: the fields of
the two record type {age :: Prelude.Double, name ::
Prelude.String} and {age :: Prelude.Double, name ::
Prelude.String, profession :: Prelude.String} must match exactly.
```

Note that the first expression returns `True` even though the fields are in different orders. The second expression signals an error because we have passed in records that have too many fields.

It is possible for a single record pattern to bind some fields to new names and other fields to punned names:

```
normalizeVectorRecord :: {#1 :: Double, #2 :: Double,
isNormalized :: Boolean} -> {#1 :: Double, #2 :: Double,
isNormalized :: Boolean};
normalizeVectorRecord vector =
    case vector of
    {#1 = x, #2 = y, isNormalized} ->
        if isNormalized then
            vector
        else
            let
                length :: Double;
                length = vectorLength (x, y);
            in
                {#1 = x / length, #2 = y / length, isNormalized =
True};
    ;
```

In the example above, we bind the ordinal fields `#1` and `#2` to the names `x` and `y` respectively. `isNormalized` is bound to its own name. With `normalizeVectorRecord` defined as above:

```
normalizeVectorRecord {#1 = 3.0, #2 = 4.0, isNormalized = False}
returns {#1 = 0.6, #2 = 0.8, isNormalized = True}
```

The final form matches records that contain *at least* the specified fields:

```
extractJob :: r\job => {r | job :: a} -> a;
extractJob record =
    case record of
    {rest | job = jobValue} -> jobValue;
    ;
```

In the example above, the `job` field of `record` is bound to the name `jobValue`. A record containing all of the fields of record *except* for `job` is bound to the name `rest`. This record is referred to as the "base record".

Note that the type of the base record (in the type signature) has a constraint. The constraint specifies that the type r does not have a field named job. This kind of constraint is referred to as a lacks constraint; see the section on Lacks constraints for details.

Punned names are permitted for both forms of non-empty record matching. In addition, any name pattern may be the wildcard name "_" (underscore) for fields that you don't want to access. The following definition of `extractJob` is equivalent to the above definition:

```
extractJob :: r\job => {r | job :: a} -> a;
extractJob record =
    case record of
    {_ | job } -> job;
    ;
```

In this version, the `job` field is bound to the name `job`, and the base record is not bound to any name (since we don't refer to it).

With `extractJob` defined as above (either definition):

```
extractJob {job = "pilot"}
returns "pilot"

extractJob {job = "telephone sanitizer", location = (3.0, 1.2)}
returns "telephone sanitizer"
```

The additional field (`location`) is ignored in the second call, because `extractJob` does not make use of the base record.

The following function does make use of the base record:

```
removeJob :: r\job => {r | job :: String} -> r;
removeJob record =
    case record of
    {rest | job = _} -> rest;
    ;
```

With `removeJob` defined as above:


```
removeJob {name = "Ford", job = "hitchhiker", age = 32.0,
location = "Earth"}
returns {name = "Ford", age = 32.0, location = "Earth"}
```

Note that a case expression that matches records can contain only a single alternative.

Matching data constructors

(See Type definitions on page 63 for an explanation of algebraic types and data constructors.)

There are three forms of pattern for matching data constructors:

```
constructor_name [identifier [identifier ...]...]
constructor_name {[field_name1 [, field_name2 ...]]}
constructor_name {field_name1 = identifier1 [, field_name2 =
identifier2 ...]}
```

The first form binds identifiers to constructor arguments in the order they appear in the declaration. i.e. the first identifier is bound to the first argument, the second identifier is bound to the second argument, etc. All arguments must be bound -- the number of identifiers must be equal to the number arguments to the data constructor. Each *identifier* may be the wildcard name "_" (underscore) for fields that you don't want to access.

The second form matches a data constructor containing *at least* the specified fields; the field name identifiers are bound to the corresponding fields' values. The third form matches a data constructor containing *at least* the specified fields; each provided identifier is matched to the values of the corresponding field.

Ex:

```
data public MyTuple a b =
    public MyTuple
        field1 :: a
        field2 :: b;

public myFst1 myTuple =
    case myTuple of
    MyTuple elem1 elem2 -> elem1;
    ;

public myFst2 myTuple =
    case myTuple of
    MyTuple elem _ -> elem;
    ;

public myFst3 myTuple =
    case myTuple of
    MyTuple {field1} -> field1;
    ;
```



```
public myFst4 myTuple =
    case myTuple of
    MyTuple {field1=elem} -> elem;
    ;
```

The above `myFst` functions extract the value of the `field1` argument from a `MyTuple` data value.

```
myFst1 (MyTuple 2.0 "String")
returns 2.0

myFst3 (MyTuple Nothing 2.0)
returns Nothing
```

Matching groups of data constructors

It is reasonably common for several case alternatives to have the same right hand side code. In this situation, the alternatives may be grouped.

As with matching data constructors, there are three forms of pattern for matching groups of data constructors:

```
(constructor_name1 [| constructor_name2 ...]) [identifier
[identifier ...]...]
(constructor_name1 [| constructor_name2 ...]) {[field_name1 [,
field_name2 ...]]}
(constructor_name1 [| constructor_name2 ...]) {field_name1 =
identifier1 [, field_name2 = identifier2 ...]}
```

The treatment of these is similar to the treatments of the corresponding forms in matching of individual data constructors. The first form binds identifiers to each constructor's arguments in the order they appear in the declaration. i.e. the first identifier is bound to the first argument, the second identifier is bound to the second argument, etc. All arguments must be bound -- the number of identifiers must be equal to the number arguments to each data constructor. Each *identifier* may be the wildcard name "_" (underscore) for fields that you don't want to access.

The second form matches data constructors containing *at least* the specified fields; the field name identifiers are bound to the corresponding fields' values. The third form matches data constructors containing *at least* the specified fields; each provided identifier is matched to the values of the corresponding field.

Ex:

```
data public MyPairOrTriple a b c =
    public MyPair
        field1 :: a
        field2 :: b |
    public MyDifferentPair
        elem1  :: a
        elem2  :: b |
    public MyTriple
        field1 :: a
        field2 :: b
        field3 :: c;

public myFirstElem myPairOrTriple =
    case myPairOrTriple of
    (MyPair | MyDifferentPair) elem _ -> elem;
    MyTriple elem _ _ -> elem;
    ;

public myField1 myPairOrTriple =
    case myPairOrTriple of
    (MyPair | MyTriple) {field1} -> field1;
    MyDifferentPair {elem1} -> elem1;
    ;
```

The above functions extract the value of the first argument from various `MyPairOrTriple` data values.

```
myFirstElem (MyPair 2.0 "Str")
returns 2.0

myField1 (MyTriple Nothing 2.0 "three")
returns Nothing
```

Matching Int values

There are two forms of pattern for matching values of type Int:

```
intValue
(intValue1 [| intValue2 ...])
```

The first form matches a single Int value. The second form matches one or more Int values.
Ex:

```
public isOneOrTwoOrMinusThree intVal =
    case intVal of
    1 -> True;
    (2 | -3) -> True;
    _ -> False;
    ;
```

With `isOneOrTwoOrMinusThree` defined as above,

```
isOneOrTwoOrMinusThree 1
returns True

isOneOrTwoOrMinusThree 4
returns False
```


Note that the pattern value is a numeric literal which is always interpreted as having type Int, rather than an ambiguous numeric type. As such, the value must fall within the constraints of the Int type.
For example, the following is invalid because the numeral is out of range:

```
public isBigNum intVal =
    case intVal of
    2147483647 -> True;
    _ -> False;
    ;
```

Matching Char values

There are two forms of pattern for matching values of type `Char`:

```
charValue
(charValue1 [| charValue2 ...])
```

The first form matches a single `Char` value. The second form matches one or more `Char` values.
Ex:

```
public isAorBorC charVal =
    case charVal of
    'a' -> True;
    ('b' | 'c') -> True;
    _ -> False;
    ;
```

With `isAorBorC` defined as above,

```
isAorBorC 'a'
returns True

isAorBorC 'd'
returns False
```

Note that the pattern value is of type `Char`, and so may be expressed using the various `Char` escape sequences.
For example, the following all test for a percent symbol:

```
public isPercent1 charVal =
    case charVal of
    '%' -> True;
    _ -> False;
    ;

public isPercent2 charVal =
    case charVal of
    '\045' -> True;
    _ -> False;
    ;
```



```
public isPercent3 charVal =
    case charVal of
    '\u0025' -> True;
    _ -> False;
    ;
```

3.2.12 Data constructor field selection

If an algebraic value is known to be a specific data constructor value, data constructor field selection may be used to access the value of that field more directly than using a case expression. It has the form:

```
expression.constructor_name.field_name
```

Ex:

```
(Just 2.0).Just.value     // The arg to Just is named 'value'
returns 2.0

(Just 2.0).Prelude.Just.value    // a qualified name is allowed
returns 2.0
```

If the wrong data constructor is encountered during field selection, a runtime error occurs:

```
(Nothing :: Maybe Double).Just.value
Error: Wrong data constructor value selected. Expecting:
Prelude.Just, found: Prelude.Nothing.
```

3.2.13 Let expressions

A let expression provides a way to introduce definitions which are local to a particular expression. It has the form:

```
let
    local_definition ;
    [local_definition ; ...]
in
    expression
```

A `local_definition` can take on one of three forms:

- A local function definition:
  `name [param1_name [param2_name ...]] = expression ;`
- A local pattern match declaration:
  `pattern = expression ;`
- A local type declaration:
  `name :: type ;`

There must be at least one local function definition or one local pattern match declaration in a let expression. The value of the let expression is the value of the expression in the body (i.e., the expression that follows "`in`").

Local function definitions

A local function definition introduces a function that is local to the let expression, and which may have zero or more parameters. For example:

```
maxSquared :: Int -> Int -> Int;
public maxSquared a b =
    let
        aSquared = square a;
        bSquared = square b;

        square x = x * x;
    in
        if aSquared > bSquared then aSquared else bSquared;
```

In the let expression above, three local functions are defined: `aSquared`, `bSquared`, and `square`. Note that variables such as `aSquared` and `bSquared`, which are declared without parameters, are considered to be local functions.

Local pattern match declarations

A local pattern match declaration allows one to bind one or more variables to the fields of a data constructor or a record using a single declaration. Such a declaration has a form:

pattern = *expression* ;

where *pattern* can have one of the following forms:

- Data constructor patterns:
  constructor_name [*name_or_wildcard* [*name_or_wildcard* ...] ...]

  or alternatively:
  constructor_name {*field_name1* [= *name_or_wildcard*] [, *field_name2* [= *name_or_wildcard*] ...]}

- List constructor patterns:
  name_or_wildcard : *name_or_wildcard*

- Tuple patterns:
  (*name_or_wildcard*, *name_or_wildcard* [, *name_or_wildcard* ...])

- Record patterns:

  Non-polymorphic record patterns:
  {*field_name1* [= *name_or_wildcard*] [, *field_name2* [= *name_or_wildcard*] ...]}

  Polymorphic record patterns:
  {_ | *field_name1* [= *name_or_wildcard*] [, *field_name2* [= *name_or_wildcard*] ...]}

In each of the above forms, `name_or_wildcard` can be either a variable or the wildcard pattern "_". Each variable appearing in the pattern is bound to the corresponding field of the expression on the right hand side.

Data constructor and list constructor patterns

Here is an example of a data constructor pattern:

```
let
    Cons {head, tail=t} = [1.0, 2.0, -3.0];
in
    (head, t)
returns (1.0, [2.0, -3.0])
```

In the above, the two fields of the `Prelude.Cons` data constructor `head` and `tail` are bound to the variables `head` and `t` respectively (the first field is a punned pattern, see Matching data constructors on page 43 for details). This let expression can also be written using the positional syntax for field bindings:

```
let
    Cons x y = [1.0, 2.0, -3.0];
in
    (x, y)
```

There is also a special syntax for pattern matching the `Prelude.Cons` data constructor with the list constructor pattern:

```
let
    x:y = [1.0, 2.0, -3.0];
in
    (x, y)
```

Record and tuple patterns

Here is an example of a record pattern:

```
let
    {country} = {country="Canada"};
    {name=_, addr=address} = {name="Zack", addr="123 Some St."};
in
    (country, address)
returns ("Canada", "123 Some St.")
```

The first declaration introduces the pattern-bound variable `country`, which is also the field being matched (this is a *punned* pattern, see Matching records on page 39 for details on punning). The second declaration uses the wildcard pattern to drop the `name` field, while binding the `addr` field to a variable called `address`.

Note that the left hand side must specify *all* the fields of the record type for the expression appearing on the right hand side. This is known as a *non-polymorphic* record pattern. This contrasts with a *polymorphic* record pattern, where this restriction is lifted:


```
let
    {_ | #2=y} = (1.0, 2.0, 3.0); // no need to specify #1, #3
    {_ | a} = {a="foo", b="bar"}; // no need to specify b
in
    (a, y)
returns ("foo", 2.0)
```

While it is possible to use a record pattern to match against a tuple (since tuples are records), it is usually more succinct to use a tuple pattern. For example,

```
let
    (x, y, z) = List.unzip3 [(1.0,0.0,0.0), (3,2,1), (6,5,4)];
in
    x ++ y ++ z
returns [1.0, 3.0, 6.0, 0.0, 2.0, 5.0, 0.0, 1.0, 4.0]
```

Lazy pattern matching

A major feature of the local pattern match declaration is its evaluation semantics: the expression on the right hand side is evaluated *only when one of the pattern-bound variables is evaluated*. In this sense, we can regard this as a form of *lazy pattern matching*.

For example, if the expression on the right hand side does not evaluate to the same data constructor as specified in the pattern, a pattern match error will *not* occur unless one of the pattern-bound variables is evaluated. Thus, the following expression will produce an error:

```
let
    Just {value} = Nothing;
in
    value :: String
Error: Wrong data constructor value selected. Expecting:
Prelude.Just, found: Prelude.Nothing.
```

However, the following expressions are okay, because the pattern-bound variables are not evaluated:

```
let
    Just {value} = Nothing;
in
    "Hello"
returns "Hello"
```

```
let
    Just {value} = undefined;
in
    "World"
returns "World"
```

The lazy evaluation semantics distinguishes this feature from case expressions, where the pattern match is attempted regardless of whether the unpacked values are needed. For example:


```
case Nothing of
Just {value} -> "Hello";
Error: Unhandled case for Prelude.Nothing.
```

```
case undefined of
Just {value} -> "World";
Error: Prelude.undefined called.
```

Local type declarations

A local type declaration can be used to provide a type signature for a local function or a pattern-bound variable appearing in a local pattern match declaration (see Type declarations on page 61 for a description of type declarations). For example, the following let expression contains a type declaration for each of the locally defined variables `x`, `y`, `z` and `square`:

```
public squareTriple triple =
    let
        x :: Double;
        y :: Double;
        z :: Double;
        (x, y, z) = triple;

        square :: Double -> Double;
        square x = x*x;
    in
        (square x, square y, square z);
```

Scope of local definitions

Take the following function for example:

```
public properFraction x =
    let
        r :: Double;
        r = x - t;

        t :: Double;
        t = truncate x;
    in
        (t, r);
```

In the above function, `t` and `r` are bound to the specified values within the let expression. In particular, a local definition is visible:

1. in its own defining expression (thus enabling recursive definitions),
2. in the defining expressions of other local definitions in the same let expression, and
3. in the body of the let expression, i.e. the expression that follows "`in`".

If there are existing definitions for the two variables `t` and `r` outside the scope of the let expression, then they are *shadowed* by the local definitions. If the shadowed definitions are top-level definitions, then they can be accessed by using qualified names; otherwise, these shadowed definitions are inaccessible in the let expression. Other definitions that are not shadowed are available both within the local definitions of the let expression and within the body.

Note also that, as with top-level definitions (see Structure of a module file on page 76), the order of the various local definitions doesn't matter. In the above example, the definition of `r` refers to `t`, and it doesn't matter that `t` is defined after `r`.

3.2.14 Lambda expressions

A lambda expression represents an anonymous function. It has the form:

```
\arg1 [arg2 ...] -> expression
```

A lambda expression creates a function accepting the specified arguments without binding it to an identifier. This function can then be passed as an argument to other functions.

Ex:

```
doubleList :: [Double] -> [Double];
public doubleList list =
    map (\x -> x*2.0) list;
```

With the above definition of `doubleList`:

```
doubleList [1.0, 8.0, 4.5]
returns [2.0, 16.0, 9.0]

doubleList []
returns []
```

3.3 Types

Every CAL value has a type. A type is a set of allowable values that a value of that type may be. For example, the `Integer` type contains all the negative and positive integers; the `Int` type contains all of the integers between -2147483648 and 2147483647; and the `Boolean` type contains only two values (`True` and `False`).

CAL types fall into a number of categories: primitive types, function types, and algebraic types. In addition, it is possible to declare foreign types. Values of foreign types are opaque to CAL, and are generally operated upon using imported foreign functions.

3.3.1 Primitive types

CAL contains a number of primitive types. The primitive types are:

Type	Example value
`Boolean`	`True, False`
`Char`	`'C', '\n', '1'`
`Byte`	`-12, 98`


```
Short       545, -12000
Int         55000, (-120000)
Float       19.9, 1.2e6
Double      16.4, 1.45e10, 0.004
Long        987000000000
String      "alpha", "\t hdr1 \n"
```

3.3.2 Built-in types

In addition to the primitive types, CAL has three built-in types with special syntactic support: lists, tuples, and records.

Lists

A list is an ordered, variable-length sequence of values of the same type. List types are specified by enclosing the type of the elements in square brackets:

```
[ type_name ]
```

For example:

```
:t ["foo", "bar"]
outputs [String]

:t [1.0, 2.0, 3.0]
outputs [Double]

:t [(1::Integer), 7, 7, 5]
outputs [Integer]
```

Note that it is possible for a list's elements to have any type, including tuples, records, and other lists:

```
:t [["foo 1", "foo 2"], [], ["bar", "baz"], ["quux"]]
outputs [[String]]

:t [("foo", (1::Integer)), ("bar", 2), ("bar", 1)]
outputs [(String, Integer)]
```

Tuples

A tuple is an ordered, constant-size collection of values of possibly differing types. Tuple types are specified by enclosing a comma-separated list of the types of each component of the tuple in parentheses:

```
( type_name, type_name [, type_name ...] )
```

For example:

```
:t (55.1, 55.2)
outputs (Double, Double)

:t (0.9, "foo", 'Z')
outputs (Double, String, Char)

:t ('C', 'A', "baz", "qux")
outputs (Char, Char, String, String)

:t ('m', ['a', 'c', 'b'])
outputs (Char, [Char])
```

```
:t (["str1", "str2"], 'c', (2.2, 1.3))
outputs ([String], Char, (Double, Double))
```

Records

A record is an unordered collection of named values of possibly differing type. Record types are specified by a list of field type specifications enclosed in braces. Field type specifications are a field name followed by a double-colon followed by the type of the field:

```
{ field1_name :: field1_type [, field2_name :: field2_type ...] }
```

Ex:

```
:t {name = "bill", job = "janitor", age = 25.0}
outputs {age :: Double, job :: String, name :: String}

:t {x = 12.7, y = (-19.0)}
outputs {x :: Double, y :: Double}

:t {a = "foo", #1 = 'N', #2 = 0.0, #3 = 1.0}
outputs {#1 :: Char, #2 :: Double, #3 :: Double, a :: String}
```

Type variables

The example algebraic types given above are extremely specific. However, it is also possible to specify much more generic types through the use of a type variable. A type variable is an identifier that "stands in" for a type in a type declaration. For example, the type signature

```
(a, a)
```

represents the type of a pair where the two components are of the same type. Type variables are distinguished from type names by the case of their initial letter. Type names always start with an upper-case letter, whereas type variables always begin with a lower-case letter. It is customary to use type variables starting from the letter a and moving up the alphabet as further variables are needed. (e.g. if one needs three type variables for a type signature, one uses the variables `a`, `b`, and `c`).

Type signature	**Matches**
`[a]`	Any list of a (for some type a)
`[[a]]`	Any list of lists of a
`(a, b, c, d)`	Any 4-tuple, each component of which may have a different type
`(a, b, b)`	Any 3-tuple whose last two components must be the same type, but whose first component may be of a different type
`(String, a)`	Any pair whose first component is a String

Note that although the type variable can represent any type, it cannot represent two different types in the same type signature: So the signature `(a, a)` can match the type `(Int, Int)` or the type `(String, String)`, but never the type `(String, Int)`.

We can define the function `appendList` as follows:

```
appendList :: [a] -> [a] -> [a];
private appendList !list1 list2 =
    case list1 of
    []     -> list2;
    head : rest -> head : appendList rest list2;
    ;
```

Note that `appendList` takes two arguments of type `[a]`. Although `[a]` can match any list type, in any given call to `appendList`, all instances of `[a]` must match the same list type:

```
appendList ['x', 'w', 'a'] ['b', 'c']
returns ['x', 'w', 'a', 'b', 'c']

appendList [[1.0, 2.0], [1.0, 3.0]] [[3.1, 3.2]]
returns [[1.0, 2.0], [1.0, 3.0], [3.1, 3.2]]

appendList [1.0, 2.0, 3.0] ['a', 'b', 'c']
Error: Type Error during an application. Caused by: Type clash:
type constructor Prelude.Double does not match Prelude.Char.
```

The third expression above fails because we are attempting to pass two lists of two different types (i.e., we are trying to simultaneously match `[a]` to `[Double]` and `[Char]`).

3.3.3 Function types

Functions are first-class values in CAL. That means that each function also belongs to a type. Function types are represented as an arrow-separated list of types. These types are the types of each of the parameters followed by the type of the return value:

```
param1_type [ -> param2_type ...] -> return_type
```

This is fairly straightforward to interpret for single-argument functions:

Type	Description
`String -> Integer`	Function from `String` to `Integer`
`Double -> Double`	Function from `Double` to `Double`

The interpretation for multi-parameter functions is slightly more subtle. The meaning of the type for a multi-parameter function hinges on the fact that the `->` operator is right-associative:

Type	Description

`Double -> Double -> String`	Function that takes 2 `Doubles` and returns a `String`
`Double -> (Double -> String)`	Equivalent to previous
`Double -> Double -> Double -> String`	Function that takes 3 `Doubles` and returns a `String`
`Double -> (Double -> (Double -> String))`	Equivalent to previous

Written in this form, it is clear that one can interpret a function taking two `Double`s and returning a string as actually being a function that accepts one `Double` and returns another function; this other function accepts another `Double` (the next parameter) and returns a string. Note how this accounts for partial function application: Applying a single argument to an *n*-argument function yields a new function of (*n*-1) arguments. *N* successive such applications eventually reduce the function call to its return value.

3.3.4 Algebraic types

An algebraic, or parameterized, type is one that allows one or more different data constructors. These data constructors may accept specified concrete types, or they may accept general types (specified by type variables).

An algebraic type is specified by a type constructor followed by zero or more other types:

```
type_constructor [type1 [type2 ...]]
```

Ex:

Type	**Possible values**
`Ordering`	`LT`, `EQ`, or `GT`
`Maybe String`	`(Just` *`string_value`*`)` or `Nothing`
`Either Int Integer`	`(Left` *`int_value`*`)` or `(Right` *`integer_value`*`)`

The last two types in the above example are subsets of the following more general types:

Type	**Possible values**
`Maybe a`	`(Just` *`value`*`)` for some *`value`*, or `Nothing`
`Either a b`	`(Left` *`value1`*`)` or `(Right` *`value2`*`)`,

	for some `value1` or `value2`

There are a number of built-in algebraic types. New ones can also be declared using a data declaration.

Type constructors vs. data constructors

Note that there is an important distinction between type constructors, which are used to specify a type, and data constructors, which are used to specify a value. Each algebraic type has exactly one type constructor, and at least one data constructor. (This distinction is occasionally obscured in types whose type constructor and data constructor have the same name).

Ex:

The type...	has type constructor...	and data constructors...
`Maybe a`	`Maybe`	`Just, Nothing`
`Either a b`	`Either`	`Left, Right`
`Ordering`	`Ordering`	`LT, EQ, GT`
`Int`	`Int`	*(int values)*

Some example values and their corresponding types:

Value	Type
`EQ`	`Ordering`
`LT`	`Ordering`
`Just "holiday"`	`Maybe String`
`Nothing`	`Maybe a` (for some `a`)
`Left 50.0`	`Either Double a` (for some `a`)

See the Standard Library Reference for a complete list of types defined in the Prelude.

3.3.5 Type classes

It is possible to organize types into groupings known as classes. A type class is a group of types that define some common set of operations. These operations are referred to as methods.

Note that in spite of the "class" and "method" terminology, CAL classes are conceptually quite different from classes in object-oriented languages such as C# or Java. Classes (and interfaces) in Java define entirely new types. By contrast, a CAL type class is not a new type itself, but rather just a set of types that share specified methods.

CAL defines a number of built-in type classes. See the Standard Library Reference for a complete list of the type classes defined in the Prelude.

Types that are members of a specific type class are said to be instances of that class. Every instance of a class specifies (in an instance declaration) the functions that are to be used to provide the functionality of the required methods. For example, the `Int` type uses the `equalsInt` function to define the functionality for the `equals` method.

See Type class definitions on page 73 and Type instance definitions on page 74 for information on declaring type classes and instances. See the Standard Library Reference for a complete list of type classes defined in the Prelude.

3.3.6 Constrained types

With type variables, it is possible to specify types extremely generally. Constrained type expressions allow one to be slightly more specific by specifying properties that type variables must satisfy, while still avoiding the need to use specific, concrete types.

There are two types of constraint that can be applied to a type expression: Type class constraints, and lacks constraints.

Constrained type signatures have two forms. Which form you use depends on whether you want to use a single constraint or multiple constraints:

```
constraint => type_signature
(constraint [, constraint [, constraint ...]]
```

It's okay to mix types of constraints in the same type signature (i.e., record constraints and type class constraints in the same signature, or even applied to the same type variable in the same signature, are allowed).

Type class constraints

Type class constraints have the following form:

```
type_class type_variable
```

One or more type variables are required to belong to various type classes. These variables are then used in the main body of the type signature. Ex:

Type	**Description**
`[a]`	List of elements of type `a`, for some type `a`
`Ord a => [a]`	List of elements of type `a`, for some type `a` which is an instance of `Ord`

`(Ord a, Show b) => a -> b`	Function taking an argument of type `a` and returning a value of type `b`, for some type `a` which is an instance of `Ord` and some type `b` which is an instance of `Show`.
`(Ord a, Show a) => (a, b)`	2-tuple whose first component is of type `a` and whose second component is of type `b`, for some type `a` which is an instance of both `Ord` and `Show` and some type `b`.

Note that the same variable can be constrained to belong to multiple classes (as in the last example above). Note also that not every variable used in the signature needs to be constrained. However, every variable that is constrained on the left of the double-arrow must be used in the signature on the right of the double-arrow.

Lacks constraints

Lacks constraints have the form:

```
type_variable \ field_name
```

A lacks constraint indicates that a type variable must be a record type which does not include the specified field. These constraints are most often used to restrict the base-record of a record type to not include the fields which will be specified for the record type (the third example in the table below):

Type	**Description**
`{r}`	A record containing any fields
`r\foo => {r}`	A record containing any fields *except* `foo`
`r\foo => {r | foo :: String}`	A record containing any fields, one of which *must be* `foo`, with `foo` having type `String`.

3.3.7 Higher-kinded type variables

CAL supports the use of *higher-kinded* type variables. Whereas a regular type variable is a variable which stands for a *type*, a higher-kinded type variable is a variable which stands for a *type constructor*. For example, one may define a functor type class as:

```
public class Functor f where
    public map :: (a -> b) -> f a -> f b;
    ;
```

In the above declaration, the variable `f` is a higher-kinded type variable standing for a type constructor with one type parameter. The type declaration of the `map` method tells us that it is a method which takes a function (of type `a -> b`) and a value of type `(f a)`, and returns a value of type `(f b)`.

The type constructor for the standard type `Maybe` fits the requirement of a type constructor with one type parameter, and thus we may define a `Functor` instance for `Maybe`:

```
instance Functor Maybe where
    map = mapMaybe;
    ;

mapMaybe :: (a -> b) -> (Maybe a -> Maybe b);
private mapMaybe mappingFunction !maybeValue =
    case maybeValue of
    Nothing -> Nothing;
    Just {value} -> Just (mappingFunction value);
    ;
```

3.4 Definitions and declarations

A definition associates a function or type with a name. This section describes the format of the various kinds of CAL definitions.

3.4.1 Function definitions

The simplest kind of definition is a function definition. A function definition has the following form:

```
[visibility] name [param1_name [param2_name ...]] = expression ;
```

Visibility is either `public`, `protected`, or `private`; if it is omitted, then the function is private by default. Public functions are visible to other modules that import the current module, whereas private functions are not. Protected functions are visible only to friend modules. (See the Modules section for further details).

Name is an identifier beginning with a lower-case letter. Zero or more parameter names may be specified.

Some example functions:


```
public e = 2.718281828459045;

public squareOf x = x * x;

private cube x = x * square x;

public distance loc1 loc2 =
    case loc1 of
    (x1,y1) -> case loc2 of
               (x2,y2) -> vectorLength (x1-x2, y1-y2);
               ;
    ;
```

Note that constant values can be defined as zero-parameter functions.

3.4.2 Type declarations

CAL uses a feature known as type inference to deduce the types for each expression and definition that it encounters. However, type inference is not always able to unambiguously determine the type of an expression:

```
:t []
outputs [a]

:t [] == []
Error: Ambiguous type signature in inferred type Prelude.Eq a =>
a.
```

In the first example above, the type inferencer is able to determine that the expression `[]` has the type `[a]`. In the second example, the inferencer is able to determine that the expression `[]` must have the type `Eq a => a` since we are attempting to apply the `equals` method of the `Eq` class to it. However, that is not sufficiently specific to determine which instance of `Eq` should be used for the `equals` method, so the attempt to type the expression fails.

In such ambiguous cases, it may be necessary to explicitly declare the type that an expression or definition is meant to have. These declarations are accomplished in CAL through a type declaration. Type declarations have the following form:

```
expression :: type ;
```

Ex:

```
pi :: Double;

sqrt :: Double -> Double;

equalsMaybe :: Eq a => Maybe a -> Maybe a -> Boolean;
```

An explicit type declaration can restrict an expression's type to a more specific type than the inferenced type, but it can never declare an expression's type to be a less specific type than the inferenced type:


```
:t []
outputs [a]

:t [] :: [Int]
outputs [Int]

:t ([] :: [Int]) == []
outputs Boolean

:t 55 + 5
outputs Num a => a

:t 55 + 5 :: Eq a => a
Error: The declared type of the expression is not compatible with
its inferred type Prelude.Num a => a. Caused by: Type clash: The
type declaration Prelude.Eq a => a does not match Prelude.Num a
=> a.
```

The final declaration fails to compile because `Num` is a subclass of `Eq`, so we are attempting to give the expression a more general type (`Eq`) than the inferenced type (`Num`).

Even when it is not necessary, it is good practice to include explicit type declarations in certain situations. For example, it is a good practice to always assert the type of functions immediately before they are defined:

```
notEqualsOrdering :: Ordering -> Ordering -> Boolean;
private notEqualsOrdering !x !y = not (equalsOrdering x y);

equalsOrdering :: Ordering -> Ordering -> Boolean;
private equalsOrdering !x !y =
    case x of
    LT ->
        case y of
        LT -> True;
        _ -> False;
        ;
    EQ ->
        case y of
        EQ -> True;
        _ -> False;
        ;
    GT ->
        case y of
        GT -> True;
        _ -> False;
        ;
    ;
```

This serves as both an important piece of documentation (ex, that the function accepts two arguments of type `Ordering` and returns a `Boolean`), and allows the compiler to provide better error messages in certain circumstances.

3.4.3 Type definitions

CAL allows the programmer to extend the type system by defining custom types. As with functions, these types may be either public, protected, or private to a module. Simple type definitions take the following form:

```
data [visibility] type_name [type_variable [type_variable ... ]]
=
   [visibility] constructor_name [arg_name :: arg_type ...]
   [ | [visibility] constructor_name [arg_name :: arg_type ...]
...]
   [deriving type_class [, type_class ...]]
   ;
```

The type is given a name and a list of one or more data constructors. The data constructors may take arguments; if they do, their names and types must be specified. The type name is an identifier starting with an uppercase letter. An argument name is either an identifier starting with a lowercase letter, or an ordinal number preceded by the number sign (#). An argument type may be either a specific type, or may be specified using type variables. If the type of a data constructor's argument is specified using a type variable, then that variable must also appear as an argument of the type constructor.

The optional `deriving` clause allows you to have the compiler automatically generate one or more type class instances for this type (see Type classes for a description of type classes). This feature saves the effort of having to define boilerplate instances for very commonly-used type classes whose instances tend to be defined in a very standard fashion.

Only certain type classes may be specified in the deriving clause of a data definition. The type classes supported are:

- `Debug.Show`
- `Prelude.Bounded`
- `Prelude.Enum`
- `Prelude.Eq`
- `Prelude.Inputable`
- `Prelude.IntEnum`
- `Prelude.Ord`
- `Prelude.Outputable`
- `QuickCheck.Arbitrary`

The type classes `Prelude.Bounded`, `Prelude.Enum`, `Prelude.IntEnum`, and `QuickCheck.Arbitrary` can only be used with the deriving clause of enumeration types, where none of the data constructors take any arguments.

Ex:


```
data public Location =
    public Nowhere |
    public Everywhere |
    public Cartesian
        x :: Double
        y :: Double |
    public Polar
        theta :: Double
        r :: Double
    deriving Eq;
```
compiles without error

```
data public Directions =
    public North | public South | public East | public West
    deriving Prelude.Eq, Prelude.Ord;
```
compiles without error

```
data public Temperature = private Hot | private Cold;
```
compiles without error

```
data public Message =
    public Warning
        message :: String |
    public Error
        message :: String ;
```
compiles without error

```
data public Tree a =
    Leaf |
    Node
        value :: a
        leftChild :: (Tree a)
        rightChild :: (Tree a);
```
compiles without error

```
data public Broken =
    public Wrong    value :: a;
```
Error: The type variable a must appear on the left-hand side of the data declaration.

The final example does not compile because a type variable (`a`) appears in a data constructor declaration but not in the type declaration.

Note that the data constructors can have a different visibility than the type itself. It is common to make types public but data constructors private when defining Abstract Data Types, for example.

3.4.4 Foreign definitions

Any Java type can be imported into CAL as a CAL foreign type.

Any Java method, constructor or field can be imported into CAL as a CAL foreign function. In addition, there are other Java operations, such as Java casts and `instanceof` operator calls that can be imported as CAL foreign functions. Although Java methods, constructors, fields, casts and `instanceof` operators all have their own special syntax in Java, they are imported into CAL uniformly as

CAL functions. They are first-class CAL functions and can be used exactly like any other CAL function.

Foreign type definitions

Java types (i.e. classes, interfaces, and Java primitive types) are imported into CAL using the `data foreign unsafe import jvm` construct:

```
data foreign unsafe import jvm [i_visibility] "java_name"
 [c_visibility] type_identifier [deriving type_class [,
type_class]];
```

The *java_name* is the fully qualified name of the type in Java. This is the name returned by the Java method `java.lang.Class.getName()`.

Some examples are:

```
"java.lang.String"

"java.io.File"
```

Java inner classes use the $ as a separator between the name of the outer class and the name of the inner class:

```
"java.util.Map$Entry"
```

Java primitive types are allowed as well:

```
"boolean", "char", "byte", "short", "int", "long", "float",
"double"
```

Java array types follow the above naming rules along with the rule that terminating square brackets are used to indicate array dimensionality (as in Java source). Some examples are:

```
"java.lang.Object[]"

"int[]"

"long[][]"

"java.lang.String[][]"
```

C_visibility specifies the visibility of the imported type; it will be either `public`, `protected`, or `private`. If omitted, the visibility defaults to `private`.

Type_identifier is a CAL identifier beginning with an uppercase letter. It specifies the name that the foreign type will be assigned within CAL.

Ex:

```
data foreign unsafe import jvm public "java.math.BigInteger"
    public Integer;
```

In the above example, the name of the type in CAL is `Prelude.Integer` (this definition occurs in the `Prelude` module). It is defined by the Java class `java.math.BigInteger`. Both the implementation visibility and the CAL visibility are public.

Ex:

```
data foreign unsafe import jvm private "int"
    public RelativeDate deriving Eq, Ord;
```

In this example, the name of the type in CAL is `RelativeTime.RelativeDate` (this definition occurs in the `RelativeTime` module). It is implemented by the Java primitive `int` type. The CAL type is public, but its implementation as a Java `int` is private. The type derives the `Eq` and `Ord` class instances.

The optional `deriving` clause allows you to have the compiler automatically generate one or more type class instances for this foreign type (see Type classes for a description of type classes). This feature saves the effort of having to define boilerplate instances for very commonly-used type classes whose instances tend to be defined in a very standard fashion.

Only certain type classes may be specified in the deriving clause of a foreign type definition. The type classes supported are:

- `Debug.Show`
- `Prelude.Eq`
- `Prelude.Inputable`
- `Prelude.Ord`
- `Prelude.Outputable`

The type class `Prelude.Ord` can only be derived for foreign types that represent either Java primitive types (e.g., `int`, `char`, `boolean`) or Java reference types that implement the `Comparable` interface.

"Implementation visibility" is specified by *i_visibility*. It will be either `public`, `protected`, or `private`. If omitted, the implementation visibility defaults to private. Implementation visibility indicates whether outside modules are permitted to define foreign functions that operate on the imported type.

This ability to control the visibility of a foreign type's implementation type allows the programmer to define abstract data types that use foreign values as their underlying representation.

Ex:


```
// --- Module Color ---
module Color;
import Cal.Core.Prelude;
friend Color_Tests;

data foreign unsafe import jvm public "java.awt.Color"
    public JColor;

data foreign unsafe import jvm protected "java.awt.Color"
    public TestableColor;

data foreign unsafe import jvm private "java.awt.Color"
    public Color;

// --- Module Color_Tests ---
module Color_Tests;
import Cal.Core.Prelude;
import Color;

// !!! Won't compile, Color has private implementation
foreign unsafe import jvm "method getRed"
    private getRed :: Color -> Int;

// OK, TestableColor is protected implementation
// and we're a friend module
foreign unsafe import jvm "method getRed"
    private t_getRed :: TestableColor -> Int;

// --- Module Draw ---
import Color using
    typeConstructor = JColor, Color;
;

// OK, JColor is public implementation
foreign unsafe import jvm "method getRed"
    private jColor_getRed :: JColor -> Int;

// !!! Won't compile, TestableColor has protected implementation
// scope and we're not a friend module
foreign unsafe import jvm "method getRed"
    private t_getRed :: TestableColor -> Int;

// !!! Won't compile, Color has private implementation scope

foreign unsafe import jvm "method getRed"
    private getRed :: Color -> Int;
```

In the above example, the `Draw` module is permitted to define foreign functions that operate upon `java.awt.Color` values as CAL functions that operate upon `JColor` values, because `JColor` is imported with a publicly-visible implementation. However, the `Draw` module is not permitted to define foreign functions that operate upon `java.awt.Color` values as CAL functions that operate upon `Color` values, because the `Color` type is defined with a private implementation. Neither can the `Draw` module define a foreign function that operates upon `java.awt.Color` values as a CAL function that operates upon `TestableColor` values, because the `Testable` type has a protected implementation. `Color_Tests` *is* able to implement such functions on `TestableColor` values, however, because `Color_Tests` is a friend module of `Color`. (See the Modules section on page 76 for more detail of scoping).

Foreign function definitions for Java methods and constructors

Java methods and constructors are both imported into CAL using the `foreign unsafe import jvm` construct. The construct takes the following form:

```
foreign unsafe import jvm "entity_type java_name"
    [visibility] function_identifier :: function_type ;
```

The valid entity types for foreign functions corresponding to Java methods and constructors are:

- `method` (followed by the Java method name)
- `static method` (followed by the Java qualified method name)
- `constructor` (optionally followed by the Java qualified type name)

Note that the type of the imported function must be specified i.e. function_type.

Here are some examples of importing non-static methods of `java.math.BigInteger`:

```
foreign unsafe import jvm "method abs"
    private absInteger :: Integer -> Integer;


foreign unsafe import jvm "method toString"
    private toStringWithRadix :: Integer -> Int -> String;
```

The `abs` method of `BigInteger` is imported as the private CAL function `absInteger`. Note in the case of the non-static method, we do not specify the fully qualified name i.e. `java.math.BigInteger.abs`. This is because the Java class in which the method is defined is determined by the type of the first argument to the method i.e. the Integer argument has Java implementation type `java.math.BigInteger`. `toStringWithRadix` corresponds to the overload of `java.math.BigInteger.toString` that takes an int argument (for the radix).

Ex:


```
foreign unsafe import jvm "static method
java.math.BigInteger.valueOf"
    public longToInteger :: Long -> Integer;
```

The static method `BigInteger.valueOf` is imported as the public CAL function `longToInteger`. For static methods, the fully qualified method name (java.math.BigInteger.valueOf) is needed.

Ex:

```
foreign unsafe import jvm "constructor"
    public stringToInteger :: String -> Integer;
```

The constructor of the `java.math.BigInteger` class that accepts a string is imported as the public CAL function `stringToInteger`. Note that optionally, the precise Java class in which the constructor is defined can be given. Otherwise, it is determined by the return type of the CAL function. For example, the following definition is equivalent:

```
foreign unsafe import jvm "constructor java.math.BigInteger"
    public stringToInteger :: String -> Integer;
```

Sometimes it is necessary to specify the javaName for a constructor, such as when the class in which the constructor is defined cannot be inferred from the return type. For example:

```
foreign unsafe import jvm "constructor java.util.ArrayList"
    private makeJList :: JList;
```

This is because CAL allows you to specify a return type whose Java implementation type is a super-type of the actual Java type returned by the Java method or constructor.

Foreign function definitions for Java fields

Java fields are imported into CAL using the `foreign unsafe import jvm` construct. The construct takes the following form:

```
foreign unsafe import jvm "entity_type java_name"
  [visibility] function_identifier :: function_type ;
```

The valid entity type for foreign fields are:

- field (followed by the Java field name)
- static field (followed by the Java qualified field name)

Note that the type of the zero-argument CAL function to be associated with the field must be specified.

Ex:


```
foreign unsafe import jvm "static field java.lang.Double.NaN"
    public notANumber :: Double;
```

Foreign function definitions for Java casts

Java casts can be imported into CAL using the `foreign unsafe import jvm` construct. You can use "cast" to convert between any two CAL types that are foreign types such that there is a legal Java conversion between the two types.

Here is an example from the `Prelude` module. In this case, the Java implementation types are `byte` and `float`. There is a legal Java cast between these two types so this declaration is allowed by CAL.

```
foreign unsafe import jvm "cast"
    byteToFloat :: Byte -> Float;
```

Here is another example from the `Prelude` module. In this case, the Java implementation types are `java.util.List` and `java.util.ArrayList`. There is a legal Java cast between these two types so this declaration is allowed by CAL.

```
foreign unsafe import jvm "cast"
    listToArrayList :: JList -> JArrayList;
```

"cast" works for any legal Java cast, including identity casts, widening and narrowing primitive casts, and widening and narrowing reference casts. The actual rules for when you can do this are somewhat technical and described in section 2.6 of the JVM specification. The CAL compiler will take care of producing optimal Java code for what you have done e.g. narrowing reference casts compile down to uses of the Java cast operator, primitive casts compile down to uses of JVM primitive conversion operations, identity and widening reference casts are no-ops.

Foreign function definitions for Java instanceof operator calls

Java `instanceof` operators can be imported into CAL using the `foreign unsafe import jvm` construct.

Here is an example from the `Prelude` module. Note that `JObject` has implementation type `java.lang.Object`:

```
foreign unsafe import jvm "instanceof java.util.Iterator"
    private isJIterator :: JObject -> Boolean;
```

This is the CAL analogue of the Java construct "`e instanceof java.lang.Iterator`" where `e` is an expression having Java static type `java.lang.Object`.

Here is another example from the Exception module. Note that `JThrowable` has implementation type `java.lang.Throwable`:

```
foreign unsafe import jvm
"instanceof java.lang.NullPointerException"
    public isJavaNullPointerException :: JThrowable -> Boolean;
```

Foreign function definitions for Java nulls and null-checks

Java provides the `null` keyword, and the JVM provides special support for comparisons of reference values to `null`. These constructs can be accessed as CAL functions as the following examples show:

```
foreign unsafe import jvm "null"
    nullString :: String;

foreign unsafe import jvm "isNull"
    isNullString :: String -> Boolean;

foreign unsafe import jvm "isNotNull"
    isNotNullString :: String -> Boolean;
```

Foreign function definitions for Java class literals

Java provides special syntax for referring to class literals (values of the type `java.lang.Class`) via the `class` keyword, and starting with Java 5, the JVM provides special support for loading class literal values. Class literals can be accessed as CAL functions as the following examples show:

```
foreign unsafe import jvm "class int"
    intClass :: JClass;

foreign unsafe import jvm "class java.util.List"
    listClass :: JClass;

foreign unsafe import jvm "class long[][]"
    longArrayArrayClass :: JClass;
```

In the above, the CAL type `JClass` has implementation type `java.lang.Class`. The three declarations are CAL analogues of the Java expressions `int.class`, `java.util.List.class`, and `long[][].class` respectively. Note that class literals can be accessed for Java primitive types (e.g. `int`), reference types (e.g. `java.util.List`), array types (e.g. `long[][]`) and also the pseudo-type `void`.

Foreign function definitions for Java array operations

Java provides primitive operator support for arrays. In particular, for creating, updating, subscripting and taking the length of an array. This functionality can be exposed as CAL functions, as the following examples show:

```
data foreign unsafe import jvm "int[]" JIntArray;

foreign unsafe import jvm "newArray"
    newIntArray :: Int -> JIntArray;

foreign unsafe import jvm "updateArray"
    updateIntArray :: JIntArray -> Int -> Int -> Int;

foreign unsafe import jvm "lengthArray"
    lengthIntArray :: JIntArray -> Int;

foreign unsafe import jvm "subscriptArray"
    subscriptIntArray :: JIntArray -> Int -> Int;
```

Note that multi-dimensional arrays are also supported, in all the variants supported by the JVM. For example, you can subscript a 2-dimensional array at one index (to get a one-dimensional array) or at 2-indices (to get an element value).

```
data foreign unsafe import jvm "int[][]"
    JInt2Array;

//specify the size of one dimension
foreign unsafe import jvm "newArray"
    newInt2Array :: Int -> JInt2Array;

//specify the sizes of both dimensions
foreign unsafe import jvm "newArray"
    newInt2Array2 :: Int -> Int -> JInt2Array;

foreign unsafe import jvm "updateArray"
    updateInt2Array :: JInt2Array -> Int -> JIntArray ->
JIntArray;

foreign unsafe import jvm "updateArray"
    updateInt2Array2 :: JInt2Array -> Int -> Int -> Int -> Int;
```



```
foreign unsafe import jvm "subscriptArray"
    subscriptInt2Array :: JInt2Array -> Int -> Int -> Int;


foreign unsafe import jvm "subscriptArray"
    subscriptInt2ArrayToIntArray :: JInt2Array -> Int ->
JIntArray;
```

3.4.5 Type class definitions

A type class is a group of types that all provide some set of operations. (See the Type classes section for details of type classes). Type classes are defined using the class keyword. Class definitions take the following form:

```
[visibility] class class_name class_variable where
    [visibility] method_name :: method_type
        [default default_implementation_function_name];
    [[visibility] method_name :: method_type
        [default default_implementation_function_name]; ...]
    ;
```

Like types, type classes can be defined as being either public, protected, or private to a module. The name of a type class is an identifier starting with an uppercase letter. Like types, the visibility specification is optional; if it is omitted, then the class defaults to being private. The class declaration is followed by a list of one or more method declarations. Each method must specify its type and can be declared as being either public or private (with private being the default). Each method may also specify a default implementation.

```
public class MyAppendable a where
    public myEmpty :: a;
    public myIsEmpty :: a -> Boolean;
    public myAppend :: a -> a -> a;
    public myConcat :: [a] -> a
        default myConcatDefault;
    ;
```

In the above example, a type `a` can be a member of the type class `MyAppendable` only if it specifies a public `myEmpty` method that returns a value of type `a`, a public `myIsEmpty` method that accepts a value of type `a` and returns a `Boolean`, a public `myAppend` method that accepts two values of type `a` and returns a value of type `a`, and a public `myConcat` method that accepts a list of elements of type `a` and returns a value of type `a`.

The `myConcat` method, which concatenates a list of values of type `a`, can be implemented using the methods `myAppend` and `myEmpty`. Thus, a default implementation can be specified, in this case through the function `myConcatDefault`:


```
myConcatDefault :: MyAppendable a => [a] -> a;
private myConcatDefault = List.foldRight myAppend myEmpty;
```

Constrained class methods

The type variable used in a type class declaration (before the `where` keyword) scopes over the entire declaration. However, one may have other type variables in the type declarations of the methods in the type class, and these type variables can have additional constraints.

Ex:

```
public class Formatter a where
    public formatBoolean :: a -> Boolean -> String;
    public formatChar :: a -> Char -> String;
    public formatNum :: Num b => a -> b -> String;
    ;
```

In the above example, the method `formatNum` has an additional type class constraint on its second argument, namely `Num b => b`.

3.4.6 Type instance definitions

Once a type class has been defined, types may be added to it by means of a type instance definition. A type instance definition adds a type to a class and specifies the functions that provide the required methods for the class. An instance declaration has the following form:

```
instance class_name type_signature where
    method_name = function_name ;
    [method_name = function_name ; ...]
    ;
```

An instance definition defines a type as being part of a type class.
Ex:

```
public class MyEq a where
    public myEquals :: a -> a -> Boolean;
    public myNotEquals :: a -> a -> Boolean;
    ;

equalsInt :: Int -> Int -> Boolean;
private equalsInt !x !y =
    x == y;

notEqualsInt :: Int -> Int -> Boolean;
private notEqualsInt !x !y =
    x != y;

// Int is an instance of MyEq
instance MyEq Int where
    myEquals = equalsInt;
    myNotEquals = notEqualsInt;
    ;
```


Note that public methods (i.e., methods that may be referenced from modules other than the one in which they were defined) may be bound to private functions.

If a method is declared in the type class to have a default implementation, then the instance declaration is allowed to omit the specification of an instance-specific version of the method.

Ex:

```
public class MyEq2 a where
    public myEquals2 :: a -> a -> Boolean;
    public myNotEquals2 :: a -> a -> Boolean
        default defaultNotMyEquals2;
    ;

defaultNotMyEquals2 :: MyEq2 a => a -> a -> Boolean;
private defaultNotMyEquals2 !x !y =
    not (myEquals2 x y);

equalsInt2 :: Int -> Int -> Boolean;
private equalsInt2 !x !y =
    x == y;

// Int is an instance of MyEq2
instance MyEq2 Int where
    myEquals2 = equalsInt2;
    ;
```

Constrained type instance definitions

There is another form of instance definition for declaring constrained types to be part of a type class:

```
instance constraints => class_name type_signature where
    method_name = function_name ;
    [ method_name = function_name ; ]
```

Ex:

```
instance Eq a => Eq (MyMaybe a) where
    equals = equalsMyMaybe;
    notEquals = notEqualsMyMaybe;
    ;


instance (Eq a, Eq b) => Eq (MyEither a b) where
    equals = equalsMyEither;
    notEquals = notEqualsMyEither;
    ;
```

The first example declares the type `MyMaybe a` to be a member of `Eq` for all types `a` that are members of `Eq`. The second example declares the type `MyEither a b` to be a member of `Eq` for all types `a` and `b` where `a` is a member of `Eq` and `b` is a member of `Eq`.

3.5 Modules

The standard compilation unit of a CAL program is the module. A module is usually stored as a text file containing CAL source.

Each definition in a CAL program resides in a single module. Private definitions can only be referenced from within the module in which they are defined. Protected definitions can be referenced from other modules only if the other module is a friend module. Public definitions may be referenced from other modules if they are imported into the other module (see Importing functions and types from other modules).

3.5.1 Module names

The name of a module has the form:

```
component[.component[.component[…]]]
```

where each `component` is an identifier starting with an uppercase letter. For example, `UserGuideExamples`, `Cal.Core.Prelude` and `Cal.Test.Core.Prelude_Tests` are all valid module names.

The ability to have multiple components in a module name allows us to organize a set of modules into a hierarchy. For example, the standard library modules listed in Section 2.6 can be viewed as forming the following hierarchy:

Cal

- Collections
  - Array, IntMap, List, LongMap, Map, Set
- Core
  - Bits, Char, Debug, Dynamic, Exception, Prelude, Record, Resource, String, System
- Utilities
  - Decimal, Locale, Math, MessageFormat, QuickCheck, Random, StringNoCase, StringProperties, TimeZone

Under this scheme, the prefix `Cal.Collections` of the module name `Cal.Collections.Array` can be considered as a namespace for the module.

To reduce the amount typing required when referring to modules with such hierarchical names, CAL permits the use of *partially qualified* module names whenever they are not ambiguous. A partially qualified module name is a proper suffix of a (fully qualified) module name. For example, the module `Cal.Collections.List` has the partially qualified names `List` and `Collections.List`.

For example, suppose we have a module:

```
module W.X.Y.Z;
import Cal.Core.Prelude;
import Y.Z;
import Z;
import A.B.C.D.E;
import P.C.D.E;
import D.E;
```

The following table lists the partially qualified and fully qualified module names of the modules involved, and to which module each name resolves.

Name	Resolves to…
`Prelude`	`Cal.Core.Prelude`
`Core.Prelude`	`Cal.Core.Prelude`
`Cal.Core.Prelude`	`Cal.Core.Prelude`
`Z`	`Z`
`Y.Z`	`Y.Z`
`X.Y.Z`	`W.X.Y.Z` (the current module)
`W.X.Y.Z`	`W.X.Y.Z`
`E`	Ambiguous (3 potential matches: `A.B.C.D.E`, `P.C.D.E`, `D.E`)
`D.E`	`D.E`
`C.D.E`	Ambiguous (2 potential matches: `A.B.C.D.E` and `P.C.D.E`)
`B.C.D.E`	`A.B.C.D.E`
`P.C.D.E`	`P.C.D.E`
`A.B.C.D.E`	`A.B.C.D.E`

The salient points in this example are:

- The fully qualified name of a module always resolves to that module.
- In the case of `C.D.E`, no preference is given to either `A.B.C.D.E` or `P.C.D.E` – it is considered ambiguous.
- Neither `Z` nor `Y.Z` resolves to the current module `W.X.Y.Z`.
- Adding an additional component to the front of a resolvable name may make it ambiguous (e.g. `D.E` → `C.D.E`)

3.5.2 Structure of a module file

A module file has the following basic structure:

```
module module_name ;

import_declaration ;
[import_declaration ; ...]

[friend_declaration ;
[friend_declaration ; ...]]

[definition ;
[definition ; ...]]
```


The initial module declaration specifies the name of the module represented by this file. The import declarations import identifiers from other modules. The friend declarations identify other modules that are allowed to import protected-scope identifiers from this module. The definitions associate identifiers with values in the current module.

Module names are identifiers that begin with an upper-case letter.

3.5.3 Importing functions and types from other modules

In order to use functions defined in another module, the other module must first be imported with an import declaration. Import declarations are of the following form:

```
import module_name [ using
    using_clause ;
    [ using_clause ; ... ] ]
;
```

Once an identifier has been imported, it can be accessed using its qualified name. A qualified name is a module name followed by a period followed by an identifier name. So for example `Prelude.String` refers to the `String` type defined in the `Prelude` module.

Ex:

```
module Example;
import Cal.Core.Prelude;

piSquare = Prelude.pi * Prelude.pi; // OK

circleArea r = pi * r * r;          // !!! Won't compile (unless
                                    //  pi is also defined in the
                                    //  Example module)
```

`using` clauses

It is often inconvenient to have to qualify the name of each imported identifier with its home module. This is particularly true for frequently-used identifiers (ex: the standard types defined in the Prelude module). As a convenience, it is possible when importing a module to specify a list of identifiers that may be referenced without qualification. The list is specified in the `using` clauses of the import statement. Each clause takes the form

```
itemKind = identifier [, identifier ...] ;
```

where *itemKind* is one of `function`, `dataConstructor`, `typeConstructor`, or `typeClass`, and the identifiers are the identifiers of that kind to import without qualification. It is possible to have multiple clauses of the same *itemKind* in a single import statement.

Ex:

```
import Cal.Core.Prelude using
    typeConstructor = Int, Double, String, Boolean, Char;
    dataConstructor = False, True, LT, EQ, GT, Nothing, Just;
    typeClass = Eq, Ord, Num, Inputable, Outputable;
    function = append, compare, concatl
    function = toDouble, field1, field2, upFrom, upFromTo;
;
import Math using
    function = pi;
;

piSquare = Math.pi * Math.pi; // Still OK

circleArea r = pi * r * r;          // OK now, we've imported pi
                                    //   as an unqualified
                                    //   identifier (i.e., in a
                                    //   using clause)
```

3.5.4 Friend modules and protected scope

It is normally only possible to import an identifier from another module if that identifier has been declared as public. Private identifiers can never be imported from other modules. However, there is a third kind of scope that identifiers can have: protected scope.

An identifier with protected scope may be imported from another module if the importing module is a friend module of the module where the protected identifier is defined. Friend modules are specified in the module where the protected identifier is defined by listing them at the top of the module after the import declarations.

Ex:

```
// Shape module
module Shape;
import Cal.Core.Prelude;
friend Shape_Tests;

drawGenericShape :: GenericShape -> String;
protected drawGenericShape genericShape =
    let
        r = unwrapShape genericShape;
    in
        r.draw r.value;
```



```
// Shape_Tests module
module Shape_Tests;
import Cal.Core.Prelude;
import Shape;

drawGeneric = Shape.drawGenericShape;  // OK, protected symbols
                                       // in Shape module are
                                       // usable from Shape_Tests
                                       // because the Shape
                                       // module declares
                                       // Shape_Tests as a friend


// ShapeConsumer module
module ShapeConsumer;
import Cal.Core.Prelude;
import Shape;

drawGeneric = Shape.drawGenericShape; // !!! Won't compile,
                                      // ShapeConsumer is not a
                                      // friend of Shape module
```

3.5.5 Workspaces

A workspace is a collection of CAL modules that are loaded at the same time and are available for import. For example, if modules Alpha, Beta, and Gamma are all in the same workspace, then it is possible for Alpha to import Beta or Gamma, but not Delta. If Alpha needs to import Delta, then Delta must be added to the workspace.

Adding modules to a workspace can be done either dynamically (by issuing commands to the CAL environment being used, such as ICE or GemCutter), or persistently, by editing the workspace specification file. The format of this file is a property of the CAL environment being used, and not of the language itself, and is therefore beyond the scope of this document.

3.6 CALDoc

A CALDoc comment is a piece of end-user and developer visible documentation in the source code. Such a comment binds exclusively to the definition that immediately follows it. CALDoc comments are allowed for the following kinds of definitions:

- modules
- functions (algebraic, foreign and primitive) and function type declarations
- type classes and class methods
- instances and instance methods
- types (algebraic and foreign) and data constructors

In the case of function definitions and function type declarations, CALDoc comments can be used to document both top-level and local definitions (i.e., those defined in `let` expressions). Also, if a function has an associated type declaration, then the comment must appear before the type declaration rather than before the function definition.

Only whitespace and regular comments may separate a CALDoc comment from its associated definition. Unbound CALDoc comments will result in compilation errors.

Here are some examples demonstrating the positioning of CALDoc comments with respect to their associated definitions:

<table>
<tr><th>Definition</th><th>Examples</th></tr>
<tr><td>Modules</td><td><pre>/** … */
module Draw;</pre></td></tr>
<tr><td>Functions and function type declarations</td><td><pre>/** … */
second list = head (tail list);
/** … */
add1 :: Int -> Int;
add1 x = add 1;

let
    /** … */
    oneTwoThree :: [Int];
    oneTwoThree = [1, 2, 3];
in
    …</pre></td></tr>
<tr><td>Type classes and class methods</td><td><pre>/** … */
public class Bounded a where
    /** … */
    public minBound :: a;
    /** … */
    public maxBound :: a;
    ;</pre></td></tr>
<tr><td>Instances and instance methods</td><td><pre>/** … */
instance Bounded Int where
    /** … */
    minBound = minBoundInt;
    /** … */
    maxBound = maxBoundInt;
    ;</pre></td></tr>
<tr><td>Types and data constructors</td><td><pre>/** … */
data public Maybe a =
    /** … */
    public Nothing |
    /** … */
    public Just
        value :: a
    deriving Eq, Ord, Inputable, Outputable;</pre></td></tr>
</table>

3.6.1 Structure of a CALDoc comment

A CALDoc comment begins with a general description section, consisting of an arbitrary block of text that may be empty, or may span one or more lines. The description can then be followed by a tag segment composed of zero or more tagged blocks.

Block tags and inline tags

A *tag* is a special marker within a CALDoc comment that the compiler is able to recognize and process. There are two kinds of tags in CALDoc: *block tags* and *inline tags*.

A *tagged block* is a section within a CALDoc comment that starts with a block tag. A block tag is formed by the '`@`' character followed by one of a few special keywords. The block extends up to, but not including, either the first line of the next tagged block, or the end of the CALDoc comment. Tagged blocks identify certain information that has a routine structure, such as the intended purpose of the arguments of a function, in a form that can be checked and processed by the compiler.

An *inline block* is a section within a CALDoc comment that is delimited by the markers '`{@`*tagName*' and '`@}`', where *tagName* is the name of an inline tag. Inline blocks can appear anywhere in a CALDoc comment where regular text can appear. Through the use of inline tags, one can create structured and formatted text for documentation purposes. For example, bulleted lists and emphasized text can be embedded into a comment via the use of the `{@unorderedList}` and the `{@em}` tag respectively.

In CAL, the use of unsupported tags will result in compilation errors.

Text and paragraphs

A block of text within a CALDoc comment is processed as a list of paragraphs. In CALDoc, two paragraphs are separated from one another by a *paragraph break* - one or more blank lines separating the two bodies of text.

There are also circumstances where a paragraph break is implied by the use of certain inline tags: for example, the end of the current paragraph is implied by the start of a list (see the section on list-related tags on page 88).

In a CALDoc comment, the '`@`' character is treated as part of a tag if it appears at the beginning of a comment line (ignoring leading asterisks and whitespace), or if it appears directly after a block tag (e.g. '`@return`'), an inline tag (e.g. '`{@em`'), or a close inline block tag '`@}`'. If you want to include the '`@`' character in your comment text in these locations, the character must be escaped as '`\@`'. As a special case, the string '`{@`' can be escaped as either '`{\@`' or '`\{@`'.

3.6.2 Supported CALDoc block tags

`@author` ***author-name***

The `@author` tag can be used in any CALDoc comment, and a CALDoc comment may contain more than one `@author` tag. The information in an `@author` block can be an arbitrary block of text. However, we recommend specifying one author per `@author` block. For example:

```
@author Luke Evans
@author Bo Ilic
```

`@deprecated` ***deprecation-notice***

The `@deprecated` tag can be used in any CALDoc comment, and it signifies that the documented definition is deprecated and is no longer recommended for use. The information in a `@deprecated` block can be an arbitrary block of text, and is meant to be a short description of why the definition is deprecated and what could be used in its place. A CALDoc comment may contain at most one `@deprecated` tag.

`@version` ***version-string***

The following is an example of a `@version` block, which may be used in any CALDoc comment:

```
@version 37.2.1-beta2
```

The version information in a `@version` block can be an arbitrary block of text, and is not verified against any predefined syntax. A CALDoc comment may contain at most one `@version` tag.

`@return` ***return-value-description***

The following is an example of a `@return` block, which may be used in CALDoc comments associated with functions and function type declarations:

```
@return {@link LT@}, {@link EQ@}, or {@link GT@} if {@code x@} is
        respectively less than, equal to, or greater than
        {@code y@}.
```

The information in a `@return` block can be an arbitrary block of text, and is meant to be a short description of the returned value. A CALDoc comment (for a function) may contain at most one `@return` tag.

`@arg` ***argument-name argument-description***

The `@arg` tag is meant to be used for documenting the arguments of functions and data constructors. As such, the `@arg` tag can be used in CALDoc comments associated with functions, function type declarations, and data constructors.

The information in an `@arg` block must consist of the name of the argument followed by a short description. The `@arg` blocks within a CALDoc comment must follow the order in which the arguments are declared in the function or data constructor definition, starting with the first `@arg` block documenting the first argument. The usual convention is that if a CALDoc comment contains any `@arg` blocks, then all arguments should be documented, one per `@arg` block.

Also, there cannot be more `@arg` tags in a CALDoc comment than the number of arguments permitted by the type of the function or data constructor. However, it is possible to have more `@arg` tags than lexically declared parameters. For example:

```
/**
 * A function application function. This function can also
 * be used in its operator form (which is $).
 *
 * @arg functionToApply the function to be applied.
 * @arg argument the argument to the function evaluation.
 * @return the result of evaluating
 *         {@code (functionToApply argument).@}
 */
apply :: (a -> b) -> a -> b;
public apply !functionToApply = functionToApply;
```

In this example, the function `apply` has only one lexically declared parameter, namely `functionToApply`. The type of the function, however, dictates that `apply` can be called with a second parameter of type `a`. Therefore, we can document this second argument, and give it a name (incidentally, the argument is called '`argument`', but it could be any other name).

This ability to document the names of unnamed function arguments also extends to foreign functions. For example:

```
/**
 * Returns the index within the specified string of the first
 * occurrence of the specified substring, starting at the
 * specified index.
 *
 * @arg stringToSearch the string to be searched.
 * @arg searchString the substring for which to search.
 * @arg fromIndex the index from which to start the search.
 * @return the index within this string of the first occurrence
 *         of the specified substring, starting at the
```



```
 *         specified index.
 */
foreign unsafe import jvm "method indexOf"
    public indexOfString :: String -> String -> Int -> Int;
```

The `@see` tag

The purpose of the `@see` tag is to indicate cross-references to other CAL definitions. It can be used in any CALDoc comment, and a CALDoc comment can contain more than one `@see` tag.

A `@see` block can take on one of the following six forms, depending on the kind of definitions to be cross-referenced:

- `@see function =` *{function reference}* `[, ` *{function reference}* `...]`
- `@see dataConstructor =` *{data constructor reference}* `[, ` *{data constructor reference}* `...]`
- `@see typeConstructor =` *{type constructor reference}* `[, ` *{type constructor reference}* `...]`
- `@see typeClass =` *{type class reference}* `[, ` *{type class reference}* `...]`
- `@see module =` *{module reference}* `[, ` *{module reference}* `...]`
- `@see` *{reference}* `[, ` *{reference}* `...]`

In each of the above `@see` block variants, a reference can either be just a name (e.g. `Eq` or `Prelude.map`), or one that is surrounded by double quotes (e.g. `"makeFileName"` or `"Debug.Show"`). Double-quoted names are not checked during the compilation process, while unquoted ones are checked to make sure that the definitions they reference do indeed exist and are found either in the current module or in its imported modules. Double-quoted names are handy for indicating cross-references to related definitions in modules that are not imported by the current module.

In all the cases above except "`@see module`", a name can either be qualified or unqualified, and is treated as a reference to a top-level definition. In particular, one cannot refer to local definitions in a `@see` block.

The last `@see` block variant above provides a handy, shorter syntax, one which omits the *context* keyword. In this variant, references of different kinds (function and class method names, module names, type constructor names, data constructor names, and type class names) can appear in the same block. One restriction with this syntax is that *unchecked*, (i.e. double-quoted) references to modules, type constructors, data constructors and type classes are not allowed in this kind of `@see` block.

The names of CAL entities are often unique enough that using this shorter syntax would suffice in many circumstances.

Also, within a `@see` block, whitespace is not important - there can be any amount of whitespace, or no whitespace at all, on either side of the '`=`' and the '`,`' separating the references.

Here are some examples of `@see` blocks:

1. From the Prelude module:

```
/**
 * Represents an ordering relationship between two values:
 * less than, equal to, or greater than.
 *
 * @see Ord
 * @see compare
 */
data public Ordering = …
```

2. A more contrived example:

```
/**
 * This is a test module that tests the compiler's CALDoc
 * handling abilities.
 *
 * @see Prelude, "Cal.Collections.List.zip", Bounded, Maybe
 * @see module = Prelude, "Cal.Collections.List", Debug
 * @see function= Prelude.map,
 *                "Cal.Collections.List.zipWith",id
 * @see typeClass=Prelude.Eq, Bounded
 * @see typeConstructor =Maybe ,"Cal.Collections.Array.Array"
 * @see dataConstructor = Prelude.Left ,
 *                        "Cal.Utilities.Locale.NoDecomposition"
 */
module CALDocTest;

import Prelude using
    function = id;
    typeConstructor = Maybe;
    typeClass = Bounded;
    ;

import Debug;
```

3.6.3 Supported CALDoc inline tags

`{@em` ***text*** **`@}`**

Displays *text* in an emphasized font. For example, in the generated HTML documentation, the text would be surrounded by a pair of `<em></em>` tags, and would normally be rendered in italics.

Note that the *text* being emphasized must not contain paragraph breaks. To emphasize text spanning more than one paragraph, use a separate `{@em}` block to surround the text on each side of the paragraph break. For example:


```
/**
 * Here is an example of emphasized text spanning more than
 * one paragraph: {@em This is a sentence in the first
 * paragraph to be emphasized.@}
 *
 * {@em This is a sentence in the second paragraph to be
 * emphasized.@}
 */
```

{@strong ***text*****@}**

Displays *text* in a strongly emphasized font. For example, in the generated HTML documentation, the text would be surrounded by a pair of `<strong></strong>` tags, and would normally be rendered in bold. The restriction on paragraph breaks for the `{@em}` tag also applies to this tag.

{@sup ***text*****@}**

Displays *text* in superscript. For example, in the generated HTML documentation, the text would be surrounded by a pair of `<sup></sup>` tags. The restriction on paragraph breaks for the `{@em}` tag also applies to this tag.

{@sub ***text*****@}**

Displays *text* in subscript. For example, in the generated HTML documentation, the text would be surrounded by a pair of `<sub></sub>` tags. The restriction on paragraph breaks for the `{@em}` tag also applies to this tag.

{@url ***url*****@}**

Inserts an inline hyperlink with *url* both as the visible text and as the target of the hyperlink. For example:

```
/**
 * Please visit {@url http://www.businessobjects.com@} for
 * more information.
 */
```

Note that no inline tags can appear within the *url* itself.

{@code ***code-text*****@}**

Displays *code-text* in a code font. Note that the whitespace in *code-text* is preserved and respected in the generated output. Also, blank lines are not interpreted as paragraph breaks within a `{@code}` block - they are simply treated as part of the whitespace content of the text.

Note that if a CALDoc comment line contains leading asterisks, it is only the whitespace to the right of such asterisks that is considered part of the code text.

`{@summary` *summary-text*`@}`

Displays *summary-text* as it would be displayed without the `{@summary}` tag, but include the text as part of the comment's summary. This tag is useful for overriding the default summary-extraction behaviour, which treats the first sentence in the first paragraph as the summary. For example, one can include more than one sentence in the summary, or have the summary extracted from some location other than the first sentence of the comment.

Note that *summary-text* cannot contain paragraph breaks. If multiple paragraphs are needed for the comment's summary, use one `{@summary}` block for each paragraph to be included in the summary:

```
/**
 * TODO: this needs more work
 *
 * {@summary This is the first sentence of the summary. This
 * is another sentence.@}
 *
 * {@summary This is the {@em second@} paragraph of the summary
 * (the third paragraph of the comment).@}
 *
 * More text of the comment...
 */
```

Note that without the use of the `{@summary}` blocks above, the summary of the comment would have been automatically determined to be "`TODO: this needs more work`".

The `{@link}` tag

Inserts an inline cross-reference to another CAL definition. This tag is very similar to the `@see` tag. The difference between the two is that `{@link}` generates a link that appears inline with the surrounding text, while `@see` places links in a separate "See Also" section.

A `{@link}` block can take on one of the following six forms, depending on the kind of definitions to be cross-referenced:

- `{@link function =` *{function reference}*`@}`
- `{@link dataConstructor =` *{data constructor reference}*`@}`
- `{@link typeConstructor =` *{type constructor reference}*`@}`
- `{@link typeClass =` *{type class reference}*`@}`
- `{@link module =` *{module reference}*`@}`
- `{@link` *{reference}*`@}`

The syntax and semantics of the references in the various variants above are the same as those for the `@see` tag. Please refer to the relevant documentation starting on page 85.

As with the `@see` tag, the names of CAL entities are often unique enough that using the shorter syntax (`{@link` *{reference}*`@}`) would suffice in many circumstances.

Within a `{@link}` block, whitespace is not important - there can be any amount of whitespace, or no whitespace at all, on either side of the '=' and before the '`@}`' ending the block.

The `{@unorderedList}`, `{@orderedList}` and `{@item}` tags

The tags `{@unorderedList}` and `{@orderedList}` respectively introduce an ordered (i.e. numbered) and an unordered (i.e. bulleted) list. These two tags serve a similar purpose to the `<ul>` and `<ol>` tags in HTML.

The appearance of an `{@unorderedList}` tag or an `{@orderedList}` tag implies the end of the preceding paragraph.

Within an `{@unorderedList}` block or an `{@orderedList}` block, the only permitted content is a set of zero or more `{@item}` blocks, which can be separated by whitespace. A `{@item}` block correspond to the `<li>` tag in HTML, and signifies a list item for the enclosing list. The content of a list item can be one or more paragraphs, separated by paragraph breaks. Also, a list can itself be nested within an item of another list. For example:

```
/**
 * Here are a few important points:
 * {@orderedList
 *    {@item This is point #1.@}
 *    {@item This is point #2.@}
 *    {@item
 *        A list item can contain more than one paragraph.
 *
 *        As shown here (this is the second paragraph).
 *    @}
 *    {@item
 *        A list item can contain any text or inline blocks,
 *        including nested lists, e.g.:
 *
 *        {@unorderedList
 *            {@item Red@}
 *            {@item Green@}
 *            {@item Blue@}
 *        @}
 *
 *        and code blocks:
```



```
 *        {@code
 *         let
 *             f = 3.0;
 *         in
 *             f + f
 *        @}
 *    @}
 * @}
 */
```

3.7 Standard functions and techniques

This section details some common techniques and functions for dealing with various common types of CAL data.

3.7.1 Lists

Lists are by far the most commonly-used algebraic data structures in CAL (and in most functional languages). There are a number of standard techniques for dealing with data stored in a list.

Higher-order functions

A higher-order function is a function that can accept another function as one of its arguments. A number of the common operations upon lists have been extracted into standard higher-order functions. These functions are defined in the `List` module.

map

The `map` function accepts a function and a list. It applies the provided one-argument function to each element of the provided list, and returns a list of the results. For example, the single expression

```
map round [1.2, 1.75, 3.0, 4.9]
returns [1, 2, 3, 5]
```

will convert a list of `Doubles` into a list of `Ints`.

filter

The `filter` function also accepts a function and a list. It applies the provided one-argument function to each element of the provided list, and returns a list of the elements for which the provided function returned `True`. Note that the provided function must be a predicate, i.e., it must return a `Boolean`.

Ex: The following expression takes a list of `Maybe`s and return only the elements that are not `Nothing`:

```
filter isJust [Just 'a', Just 'b', Nothing, Nothing, Just 'c']
returns [Just 'a', Just 'b', Just 'c']
```


This can be combined with `map` to obtain a list of the `Char`s that are contained in the non-`Nothing` elements:

```
map fromJust (filter isJust [Just 'a', Just 'b', Nothing,
Nothing, Just 'c'])
returns ['a', 'b', 'c']
```

foldRight and foldLeftStrict

`foldRight` accepts a 2-argument function (called the "folding function"), an initial value, and a list. The folding function is applied in turn to each element of the list along with the result of the previous application (or with the starting value, for the first application). The result of all of these applications is a single result value.

In other words, `foldRight` returns the result of applying the folding function to the first element of the list argument and the result of a recursive call to `foldRight`. Ex:

```
foldRight add 0.0 [5, 4, 6, 1]
```

is equivalent to

```
add 5 (add 4 (add 6 (add 1 0.0)))
```

There is a related function called `foldLeft`, which returns the result of applying its function argument to a recursive call to `foldLeft` and the first element of the list argument. Ex:

```
foldLeft add 0.0 [5, 4, 6, 1]
```

is equivalent to

```
add (add (add (add 0.0 5) 4) 6) 1
```

In most situations, `foldLeft` is less space-efficient than `foldRight` and should not be used. The reason has to do with differences in how the calls are reduced. `foldRight` reduces to a call to the folding function (with one argument being a recursive call to `foldRight`), whereas `foldLeft` reduces to another call to `foldLeft`. Ex:

```
foldRight add 0.0 [5, 4, 6, 1]
```

reduces (after a single reduction step) to

```
add 5 (foldRight add 0.0 [4, 6, 1])
```

In contrast, the following `foldLeft` call

```
foldLeft add 0.0 [5, 4, 6, 1]
```

reduces (after a single reduction step) to

```
foldLeft add (add 0.0 5) [4, 6, 1]
```

Because the `foldRight` call reduces to a call to the folding function, the folding function has the opportunity to begin producing output before the entire list is evaluated. In contrast, when using `foldLeft`, the folding function will not be evaluated until the entire list has been traversed. This makes `foldLeft` particularly ill-suited to processing infinite lists.

However, there is a strict version of `foldLeft` called `foldLeftStrict` which allows for a more efficient evaluation of functions that are strict in both arguments. This is because `foldLeftStrict` evaluates the call to the function argument at each recursive step, whereas `foldRight` returns a lazy value representing the application of the function argument to an element of the list argument and a recursive call to `foldRight`. So, the following call

```
foldLeftStrict add 0.0 [5, 4, 6, 1]
```

reduces (after a single reduction step) to

```
foldLeftStrict add 5.0 [4, 6, 1]
```

because the call to the folding function is evaluated immediately. This is a much more efficient way to fold a strict function over a list.

Fortunately, out of all this analysis comes a simple rule of thumb: When the folding function is strict in both arguments, use `foldLeftStrict`. Otherwise, use `foldRight`.

Some functions, such as `subtract`, are not commutative. That is, the order of their arguments makes a difference. For such functions, `foldLeftStrict` and `foldRight` can produce different results. Ex:

```
foldLeftStrict subtract 0.0 [1, 1, 1]
returns -3.0

foldRight subtract 0.0 [1, 1, 1]
returns 1.0
```

In these situations, it's necessary to choose the function that produces the correct associativity. If you need a function with left associativity, it is better to choose `foldLeftStrict` than `foldLeft` if at all possible. Only choose `foldLeft` if you need left-associativity and it is important that the folding function's arguments be evaluated non-strictly.

List-manipulation utility functions

In addition to the standard higher-order functions for operating on lists, CAL provides some utility functions for accessing elements and properties of a list.

head and tail

The `head` function accepts a list and returns its first element. The `tail` function accepts a list and returns all of the list's elements except the first. Ex:

```
head [1,2,3]
returns 1
```



```
tail [1,2,3]
returns [2,3]
```

length

The `length` function accepts a list and returns its length. The empty list has length 0, a list with one element has length 1, etc.

Recursive list handling

A common pattern is to process the elements of a list one at a time, unpacking the head and tail of the list using a `case` expression:

```
sumList1 :: Num a => [a] -> a;
public sumList1 list =
    let
        sumHelper list !acc =
            case list of
            [] -> 0;
            head : rest -> sumHelper rest (acc + head);
            ;
    in
        sumHelper list 0;
```

Note that many such functions can be replaced by calls to `foldRight` or `foldLeftStrict` (when a single value is being accumulated) or to `map` or `filter` (when a list is being transformed into another list). Ex:

```
sumList :: Num a => [a] -> a;
public sumList list = foldLeftStrict add 0 list;
```

3.7.2 Records

Unpacking named fields using the field selection operator

The easiest way to extract a specific field from a record is by using the field selection operator (`.`):

```
{name = "Phillipe", age = 5.0}.age
returns 5.0
```

Attempting to retrieve a field from a record that does not contain that field results in an error:

```
{name = "Phillipe", age = 5.0}.job
Error: Type error. Invalid record selection for field job. Caused
by: the record type {age :: Prelude.Double, name ::
Prelude.String} is missing the fields [job] from the record type
a\job => {a | job :: b}.
```

Unpacking named fields using case expressions

Another method for handling records in CAL is the case expression (see Matching records on page 39 for a detailed description). The usual situation is to want to extract a specific named field or fields from a record, which may contain other, unspecified fields:


```
recTo2DPoint rec =
    case rec of
    {_ | x, y} -> (x,y);
    ;
```

3.7.3 Tuples

Unpacking tuple components using the field selection operator

Since tuples are special cases of records, the same field selection operator that is used to extract named fields from records can be used for tuples. The first component of a tuple is named `#1`, the second component is named `#2`, etc:

```
('C', 0.61, 0.65).#1
returns 'C'
```

As with general records, attempting to extract a component that a tuple doesn't contain will result in a runtime error:

```
('C', 0.61, 0.65).#4
Error: Type error. Invalid record selection for field #4. Caused
by: the record type (Prelude.Char, Prelude.Double,
Prelude.Double) is missing the fields [#4] from the record type
a\#4 => {a | #4 :: b}.
```

Unpacking tuple components using case expressions

See Matching tuples on page 39 for a detailed description of using case expressions to extract components from tuples. Case expressions allow an expression to refer to each component of a tuple by name:

```
twoDPointToRec point =
    case point of
    (xVal, yVal) -> {x=xVal, y=yVal};
    ;
```

Unpacking tuple components using utility functions

CAL also provides a number of utility functions for extracting the components of various tuple types. These functions have names of the form `fieldN`, and return the *N*th component of a tuple. Ex:

```
field1 (1,2,3)
returns 1

field2 (1,2,3)
returns 2

field3 (1,2,3,4)
returns 3

field3 (1,2,3,4,5)
returns 3
```

These functions are provided for the first 7 components (i.e., `field1` to `field7`).

In addition, two specially-named functions are provided for the first two components:

```
fst (1,2)
returns 1

snd (1,2)
returns 2

snd (1,2,3)
Error: Type Error during an application. Caused by: the fields of
the two record type (a, b) and (Prelude.Double, Prelude.Double,
Prelude.Double) must match exactly.
```

Unlike `field1` and `field2` (which can be applied to any record with a `#1` or `#2` field respectively), `fst` and `snd` can only be applied to pairs. This is why the third expression above fails.

3.7.4 Algebraic types

Unpacking data constructor using case expressions

See Matching data constructors on page 43 for a detailed description of using case expressions to unpack data constructors. Case expressions are the most general way to extract components from an algebraic value:

```
data MyTupleType a =
    MyTuple2
        elem1 :: a
        elem2 :: a |
    MyTuple3
        elem1 :: a
        elem2 :: a
        elem3 :: a
    ;

addMyTupleElems :: Num a => MyTupleType a -> a;
addMyTupleElems myTuple =
    case myTuple of
    MyTuple2 elem1 elem2 -> elem1 + elem2;
    MyTuple3 elem1 elem2 elem3 -> elem1 + elem2 + elem3;
    ;
```

Extracting data constructor arguments using the field selection operator

If an algebraic value is known to be a specific data constructor value, and that value has only one component field of interest, the easiest way to extract that value is by using the field selection operator (.):

```
(Just 2.0).Just.value     // The arg to Just is named 'value'
returns 2.0
```

If the wrong data constructor is encountered during field selection, a runtime error occurs:


```
(Nothing :: Maybe Double).Just.value
Error: Wrong data constructor value selected. Expecting:
Prelude.Just, found: Prelude.Nothing.
```

3.8 Advanced topics

3.8.1 Evaluation of expressions

Most nested expressions can be evaluated in multiple orders. For example, with increment defined as follows:

```
increment :: Int -> Int;
increment x = x + 1;
```

consider the expression `increment (2 * 3)`. One possible order to evaluate this expression is:

```
increment (2 * 3)
(2 * 3) + 1
6 + 1
7
```

Another is:

```
increment (2 * 3)
increment 6
6 + 1
7
```

In the first case, we chose to reduce the application of the `increment` function first. In the second case, we chose to reduce the application of the `*` operator first.

We say that in the first case, we chose to reduce the "outermost reducible expression" (or outermost redex) first. An outmost redex is one that is contained in no other redex. In the second case, we chose to reduce the innermost redex first. An innermost redex is one that contains no other redex.

Let's look at another example. First, the outermost-first reduction:

```
head (45 : list1 (myFactorial 3))
45
```

And the same expression evaluated using innermost-first reduction:


```
head (45 : list1 (myFactorial 3))
head (45 : list1 (3 * (myFactorial (3 - 1)))
head (45 : list1 (3 * (myFactorial 2))
head (45 : list1 (3 * 2 * (myFactorial (2 - 1)))
head (45 : list1 (3 * 2 * (myFactorial 1))
head (45 : list1 (3 * 2 * 1 * (myFactorial (1 - 1)))
head (45 : list1 (3 * 2 * 1 * (myFactorial 0))
head (45 : list1 (3 * 2 * 1 * 1))
head (45 : list1 (6 * 1 * 1))
head (45 : list1 (6 * 1))
head (45 : list1 6)
head (45 : [6])
45
```

Note that the innermost-first reduction order requires many more steps to calculate the same result. This is because it calculates the value of the `list1 (myFactorial 3)` expression - A value that is ultimately discarded.

CAL uses an outermost-first, or "lazy", evaluation order. This contrasts with most languages, which use an innermost-first, or "strict", evaluation order. Lazy evaluation gets its name because it avoids calculating the values of expressions that are not needed for the final value of an expression (such as `list1 (myFactorial 3)`).

We can verify that CAL uses lazy evaluation by entering the following expression into ICE:

```
head (45 : list1 (error "you should never see this")) :: Int
returns 45
```

If the application of `error` had been evaluated (as it would have been in a strict evaluation order), an error message would have been displayed. Instead, a value of 45 was returned.

Infinite data structures

One of the techniques that lazy evaluation makes possible is the infinite data structure. For example, it is possible to write a function in CAL that returns a list of all of the non-negative even numbers:

```
evensFrom :: Int -> [Int];
evensFrom start =
    start : evensFrom (start + 2);

evens :: [Int];
evens = evensFrom 0;
```

In a non-lazy (or "strict") language, any application of the `evens` function would result in an infinite loop, with the computer attempting to calculate the infinite list of every even number before it could return. However, in CAL, it is possible to safely apply `evens`, so long as only some finite subset of the list is required. For example, using the `evens` function defined above in conjunction with the standard `take` function (which returns the first *n* elements of any list, for some *n*), we can find the first 5 even numbers:

```
take 5 evens
returns [0, 2, 4, 6, 8]
```

The `seq` function

There are some situations where lazy evaluation is undesirable. Strict evaluation can be both faster and more space-efficient than lazy evaluation for non-shared values that are actually evaluated.

For situations where this is known to be the case, CAL provides the ability to force the order in which expressions will be reduced. One way to do this is by using the `seq` function. The `seq` function takes two arguments. It forces the first argument to be reduced until it is in "Weak Head Normal Form"; then it returns the value of the second argument.

For algebraic types (i.e., lists, records, tuples, and user-defined types), Weak Head Normal Form is the point where the outermost data constructor for the value is known. For numeric types (`Double`, `Int`, et al), Weak Head Normal Form is the point at which the numeric value of the expression is known.[5]

Note the distinction between "evaluated until WHNF" and "evaluated completely". Lists are evaluated only until the outermost data constructor is known (i.e., until it is known whether the list is empty or not). This means that it is still possible to force the evaluation of infinite lists using `seq` without causing an infinite loop.

As an example of the effect of the seq function, consider the following two definitions:

```
cons :: a -> [a] -> [a];
cons x y = x : y;

strictCons :: a -> [a] -> [a];
strictCons x y = seq y (x : y);
```

Both `cons` and `strictCons` return a list containing the first argument followed by the elements of the second argument. However, their reductions look quite different. Here is the reduction of `cons (myFactorial 3) (cons (3 + 4) [8, 9])`:

[5] One can think of a numeric type as an algebraic type with an infinite number of data constructors. e.g., one can think of the Integer type as being defined by `data public Integer = 0 | 1 | 2 | 3 | ...`. When one thinks of numeric types in this way, then the two definitions of Weak Head Normal Form are equivalent.


```
cons (myFactorial 3) (cons (3 + 4) [8, 9])
(myFactorial 3) : (cons (3 + 4) [8, 9])
(3 * (myFactorial (3 - 1)) : (cons (3 + 4) [8, 9])
(3 * (myFactorial 2) : (cons (3 + 4) [8, 9])
(3 * 2 * (myFactorial (2 - 1)) : (cons (3 + 4) [8, 9])
(6 * (myFactorial (2 - 1)) : (cons (3 + 4) [8, 9])
(6 * (myFactorial 1)) : (cons (3 + 4) [8, 9])
(6 * 1 * (myFactorial (1 - 1))) : (cons (3 + 4) [8, 9])
(6 * (myFactorial (1 - 1))) : (cons (3 + 4) [8, 9])
(6 * (myFactorial 0)) : (cons (3 + 4) [8, 9])
(6 * 1) : (cons (3 + 4) [8, 9])
6 : (cons (3 + 4) [8, 9])
6 : ((3 + 4) : [8, 9])
6 : (7 : [8, 9])
[6, 7, 8, 9][6]
```

Compare this with the reduction of the equivalently-valued `strictCons (myFactorial 3) (strictCons (3 + 4) [8, 9])`:

```
strictCons (myFactorial 3) (strictCons (3 + 4) [8, 9])
strictCons (myFactorial 3) ((3 + 4) : [8, 9])
strictCons (myFactorial 3) (7 : [8, 9])
strictCons (myFactorial 3) [7, 8, 9][7]
(myFactorial 3) : [7, 8, 9]
(3 * myFactorial (3 - 1)) : [7, 8, 9]
(3 * myFactorial 2) : [7, 8, 9]
(3 * 2 * myFactorial (2 - 1)) : [7, 8, 9]
(6 * myFactorial (2 - 1)) : [7, 8, 9]
(6 * myFactorial 1) : [7, 8, 9]
(6 * 1 * myFactorial (1 - 1)) : [7, 8, 9]
(6 * myFactorial (1 - 1)) : [7, 8, 9]
(6 * myFactorial 0) : [7, 8, 9]
(6 * 1) : [7, 8, 9]
6 : [7, 8, 9]
[6, 7, 8, 9][8]
```

Note that `strictCons`'s use of the `seq` function forces its second argument (in this case, `(strictCons (3 + 4) [8, 9])`) to be reduced before its first argument (`myFactorial 3`).

Strict parameters

CAL provides the ability to flag certain parameters as strict by prepending them with an exclamation point (or "pling"):

```
myHead :: [a] -> a;
public myHead !list =
    case list of
    firstElement : _  -> firstElement;
    []      -> error "empty list.";
    ;
```

[6] This step is just a rewrite for clarity, not a reduction. `[6, 7, 8, 9]` is a shorthand notation for `6 : (7 : (8 : ( 9 : [])))`

[7] See footnote 6

[8] See footnote 6

In the definition of `myHead` above, for example, the `list` parameter has been flagged as strict. This indicates that the value passed into head as `list` will be evaluated until it is in "Weak Head Normal Form" before the function itself is evaluated.

Plinged arguments are evaluated in left-to-right order before the body of the function is returned. For example, the following two definitions of `notEqualsList` are equivalent:

```
private notEqualsList1 !l1 !l2 =
    not (equalsList l1 l2);

private notEqualsList2 l1 l2 =
    seq l1
    (seq l2
    (not (equalsList l1 l2)));
```

The first version flags the `l1` and `l2` parameters as strict. The second version uses nested applications of `seq` to force `l1` and l2 to be reduced (in that order) before the body expression is returned.

It is good practice to flag a parameter to a function as strict whenever you know that its value (or data constructor) will be required for the evaluation of the function. Consider the following definition of the Boolean `and` operation:

```
and :: Boolean -> Boolean -> Boolean;
public and !a b =
    case a of
    False -> False;
    True -> b;
    ;
```

Note that the first parameter is strict, but the second is not. This is because the data constructor of the first parameter is required in order to determine which case alternative to follow. However, the value of the second parameter does not influence the execution of the function. It is returned as-is in the `True` alternative, and not at all in the `False` alternative. Since the value of the second parameter is not needed in order to determine the result of the `and` function, there is no reason to flag it as strict.

Recursion and the stack

In most languages, each function call requires additional stack space. This means that the number of recursive function calls that can be made by a function in these languages is limited by the size of the stack.

However, thanks to lazy evaluation, recursive CAL functions often use no more stack than a single function call. For example, consider this definition of a function similar to the `List.map` function:


```
/**
 * 'myMap mapFunction list' applies the function mapFunction to
 * each element of the list and returns the resulting list.
 *
 * @arg mapFunction
 *       a function to be applied to each element of the list.
 * @arg list
 *       the list.
 * @return the list obtained by applying mapFunction to each
 * element of the list.
 */
myMap :: (a -> b) -> [a] -> [b];
public myMap mapFunction !list =
    case list of
    [] -> [];
    listHead : listTail ->
        mapFunction listHead : myMap mapFunction listTail;
    ;
```

Each call to `myMap` returns either `[]` (when the empty list is passed in), or a `Cons` data constructor whose head is a (not yet evaluated) application of `mapFunction` to `listHead` and whose tail is a (not yet evaluated) application of `myMap` to `mapFunction` and `listTail`. In other words, each step of the reduction of an expression involving `myMap` is a single function call, rather than being a list of nested function calls as it would be in an eager language such as Java or C++.

There is, however, one class of recursive CAL function that does use additional stack for each recursive call. Any function whose return value is strict (in the sense that it is either a numeric value or a type whose data constructors take strict arguments) may require additional stack for each recursive call.

Ex:

```
data MyLazyList a =
    MyLazyNil |
    MyLazyCons
        head :: a
        tail :: (MyLazyList a);

data MyStrictList a =
    MyStrictNil |
    MyStrictCons
        head :: a
        tail :: !(MyStrictList a);

makeMyLazyList :: [a] -> MyLazyList a;
makeMyLazyList x =
    case x of
    [] -> MyLazyNil;
    first : rest -> MyLazyCons first (makeMyLazyList rest);
    ;
```



```
makeMyStrictList :: [a] -> MyStrictList a;
makeMyStrictList x =
    case x of
    [] -> MyStrictNil;
    first : rest -> MyStrictCons first (makeMyStrictList rest);
    ;
```

In the example code above, we have defined two types that have the same form as the built-in `List` type: `MyLazyList` and `MyStrictList`. `MyLazyList` is equivalent to the built-in `List` type (with the `MyLazyNil` data constructor corresponding to the `Nil` data constructor, and the `MyLazyCons` data constructor corresponding to the `Cons` data constructor). `MyStrictList` is equivalent to the built-in `List` type with one difference: The `tail` parameter of `MyStrictCons` is strict rather than lazy.

The `makeMyStrictList` and `makeMyLazyList` functions convert a regular CAL list into a `MyStrictList` and `MyLazyList` respectively. However, one causes a stack overflow when applied to large lists, while the other does not:

```
(makeMyLazyList (upFromTo 0 100000 :: [Int])).MyLazyCons.head
returns 0

(makeMyStrictList (upFromTo 0 100000 :: [Int])).MyStrictCons.head
Error: The java virtual machine encountered an error.
Caused by: java.lang.StackOverflowError,  Detail: null
```

The reason that `makeMyStrictList` requires additional stack for each call is that in order to evaluate the first call, it must evaluate all of the recursive calls as well (since the strict `tail` parameter requires the recursive call to be reduced to WHNF before the `MyStrictCons` value can be constructed). By contrast, `makeMyLazyList` can construct and return the `MyLazyCons` value without having to make a recursive call. The recursive call will happen if and when the value of `tail` is requested.

Tail recursive functions

Even recursive functions that must return strict values can be written to require only constant stack, so long as they are tail recursive.

A tail recursive function is a recursive function where the recursive call (or calls) is in tail position. A function call is in tail position if the result of the function call is returned directly:

```
sumIntList1 :: [Integer] -> Integer;
sumIntList1 numList =
    case numList of
    [] -> 0;
    first : rest -> first + sumIntList1 rest;
    ;
```



```
sumIntList2 :: [Integer] -> Integer -> Integer;
sumIntList2 numList accum =
    case numList of
    [] -> accum;
    first : rest -> sumIntList2 rest (accum + first);
    ;
```

In the above code, the recursive call to `sumIntList2` is in tail position, because the result from `sumIntList2` is returned directly without any additional processing.

In contrast, the recursive call to `sumIntList1` is *not* in tail position, because the result from the call to `sumIntList1` is passed as an argument to the addition operator rather than being returned directly. The addition operator is in tail position in `sumIntList1` rather than the recursive call.

Although `sumIntList2` does not use extra stack at each recursive invocation, it is still not ideal. Its `accum` parameter is lazily evaluated. This means that an invocation such as `sumIntList2 [5,6,7,8] 0` would reduce as follows:

```
sumIntList2 [5, 6, 7, 8] 0
sumIntList2 [6, 7, 8] (5 + 0)
sumIntList2 [7, 8] (6 + (5 + 0))
sumIntList2 [8] (7 + (6 + (5 + 0)))
sumIntList2 [] (8 + (7 + (6 + (5 + 0))))
(8 + (7 + (6 + (0 + 5))))
```

Although the recursive invocations of `sumIntList2` took no extra stack, the invocations of the addition operator to evaluate `accum` at the end of the recursion will require enough stack to make *n* calls (where the input list had *n* elements). This means that the tail-recursive `sumIntList2` can still overflow the stack for sufficiently large inputs!

Fortunately, by making the `accum` parameter strict, we can force each addition to be evaluated at the time of the recursive call. This eliminates the need to evaluate a large, nested addition at the end of the recursion.

```
sumIntList3 :: [Integer] -> Integer -> Integer;
sumIntList3 numList !accum =
    case numList of
    [] -> accum;
    first : rest -> sumIntList3 rest (accum + first);
    ;
```

`sumIntList3` is identical to `sumIntList2`, except that its accumulator argument is flagged as strict. This forces its accum argument to be reduced before each recursive application:


```
sumIntList3 [5, 6, 7, 8] 0
sumIntList3 [6, 7, 8] (5 + 0)
sumIntList3 [6, 7, 8] 5
sumIntList3 [7, 8] (6 + 5)
sumIntList3 [7, 8] 11
sumIntList3 [8] (7 + 11)
sumIntList3 [8] 18
sumIntList3 [] (8 + 18)
sumIntList3 [] 26
26
```

Note that at each stage of reduction, the value of accum is always an expression with 0 or 1 addition operations. Unlike the previous two sum functions, this strict tail-recursive version can be called with a list of any size without overflowing the stack:

```
sumIntList1 (upFromTo 0 5000)
Error while executing: The java virtual machine encountered an error.
Caused by: java.lang.StackOverflowError,  Detail: null

sumIntList2 (upFromTo 0 5000) 0
Error while executing: The java virtual machine encountered an error.
Caused by: java.lang.StackOverflowError,  Detail: null

sumIntList3 (upFromTo 0 20000) 0
returns 200010000

sumIntList3 (upFromTo 0 100000) 0
returns 5000050000
```

3.8.2 Dynamic typing

Most of the time, CAL's type system provides an extremely valuable mechanism for helping to ensure program correctness. However, there are rare occasions where you want to have, for example, a list that can contain different types of data. CAL's type system does not normally allow lists whose elements are not all of the same type. However, by wrapping each value (of whichever type) in a `Dynamic` value, it is possible to get around this restriction:

```
processSomeDynamicValues :: Dynamic -> String;
processSomeDynamicValues v =
    let
        intValue = fromDynamicWithDefault v (0 :: Int);
        stringValue = fromDynamicWithDefault v "";
        doubleValue = fromDynamicWithDefault v 0.0;
    in
        if (intValue != 0) then
            intToString intValue
        else if (stringValue != "") then
            stringValue
        else if (doubleValue != 0) then
            doubleToString doubleValue
        else
            "unknown Dynamic value";
```


With `processSomeDynamicValues` defined as above, we can evaluate the following expressions:

```
processSomeDynamicValues (toDynamic (35 :: Int))
returns "35"

map processSomeDynamicValues [toDynamic (16 :: Int), toDynamic
"str", toDynamic 1.2, toDynamic 18.0, toDynamic (Just 'C')]
returns ["16", "str", "1.2", "18.0", "unknown Dynamic value"]
```

The `Dynamic` module contains a number of functions for creating and inspecting `Dynamic` values. The `toDynamic` function is used to create `Dynamic` values. It accepts an argument of any `Typeable` type[9], and returns a `Dynamic` value that wraps that type.

The `fromDynamicWithDefault` function is used to extract values from a `Dynamic` value. It takes two arguments: a `Dynamic` value, and a default value. It returns the value wrapped by the `Dynamic` value if the wrapped value is of the same type as the default value. If they are not of the same type, then it returns the default value.

The other way to extract values from a `Dynamic` wrapper is using the function `fromDynamic`. It returns a value of `Just v` if the `Dynamic` value wraps the "expected" type, or `Nothing` otherwise:

```
unwrapInt :: Dynamic -> Maybe Int;
unwrapInt v =
    fromDynamic v;
```

With `unwrapInt` defined as above, we can evaluate the following expressions:

```
unwrapInt (toDynamic (50 :: Int))
returns Just 50

unwrapInt (toDynamic 50.0)
returns Nothing
```

Note that `unwrapInt` disambiguates the "expected" type by declaring the return type of `unwrapInt`. When `fromDynamic` is called, there must always be some way for the compiler to determine the expected return type:

```
fromDynamic (toDynamic (50 :: Int)) :: Maybe Int
returns Just 50

fromDynamic (toDynamic 50.0) :: Maybe Int
returns Nothing

fromDynamic (toDynamic 50.0) :: Maybe Double
returns Just 50.0

fromDynamic (toDynamic 50.0)
Error: Ambiguous type signature in inferred type
(Prelude.Outputable a, Prelude.Typeable a) => a.
```

[9] All non-polymorphic CAL types, and all polymorphic types whose type argument variables are not higher-kinded are automatically instances of the `Typeable` class.

The final expression fails, because the type inferencer has not been given enough information to determine the expected type, and therefore whether the result should be `Just 50.0` (if `Maybe Double` is the expected type) or `Nothing` (if some other type is expected).

3.8.3 CAFs and caching

A Constant Applicative Form, or CAF, is a top-level non-foreign function that has no lexical arguments and has a non-constrained type.

```
approxPi :: Double;
approxPi = 3.14159;

addOneInt :: Int -> Int;
addOneInt x = x + 1;

sunDiameter :: Num a => a;
sunDiameter = 1380000;
```

Of the three top-level functions above, only `approxPi` is a CAF. `addOneInt` is not a CAF, because its definition includes a lexical argument (`x`). `sunDiameter` is also not a CAF, because it has a constrained type[10] (i.e., its type signature contains a '`=>`').

Even though CAFs must not accept lexical arguments, it is nevertheless possible to declare a CAF which equals a function of one or more arguments:

```
cafAdd10Int :: Int -> Int;
cafAdd10Int = add 10;

add10Int :: Int -> Int;
add10Int x = add 10 x;
```

In the code above, `add10Int` and `cafAdd10Int` have the same type and represent functions with equivalent behaviors (viz., they both add 10 to their single argument). However, `cafAdd10Int` is a CAF, whereas `add10Int` is not, because `add10Int` accepts a *lexical* argument (i.e., one which is specified in the function definition).

Note also the distinction between polymorphic types and constrained types. A CAF may have a polymorphic type, so long as the polymorphic type is unconstrained:

```
second :: [a] -> a;
second = head `compose` tail;

secondOrderable :: Ord a => [a] -> a;
secondOrderable = head `compose` tail;
```

[10] Functions with a constrained type are not CAFs because their underlying representation accepts a hidden argument that indicates what specific type they should take in each given context.

In the above code, second is a CAF, whereas `secondOrderable` is not, since `secondOrderable` has a class constraint.

CAL caches the value of each CAF after the first time it has been evaluated. This means that it is possible to use the value of a expensive-to-calculate CAF in multiple expressions (or in multiple parts of a single expression) without incurring the cost of evaluating it multiple times.

```
largeSum :: Integer;
largeSum = sum (take 1000000 (List.repeat 1));
```

We can verify this using a (somewhat contrived) example. With `largeSum` defined as above, in a module in the current workspace:

```
largeSum
returns 1000000

largeSum
returns 1000000
```

The first time that we evaluate `largeSum` in an ICE session, it takes an appreciable amount of time to execute (on one machine it took 10,250 milliseconds). Subsequent times, however, it should evaluate nearly instantaneously (0 milliseconds on the same machine).

In the case of CAFs that represent function values (such as `cafAdd10Int` above), it is important to note that the value which is cached is not the value of any particular invocation (e.g., the value of `16` for `cafAdd10Int 6`), but rather the value of the function itself.

Because the values of CAFs are always cached, it is important to be aware that CAFs that evaluate to large values can significantly increase the memory requirements of your program.

4 Standard Library Reference

4.1 Types

4.1.1 Dynamic

The `Dynamic` type (provided by the `Dynamic` module) is used for holding values of other CAL types.

`Dynamic` values are created using the `toDynamic` function. They are extracted again using either the `fromDynamicWithDefault` function, or the `fromDynamic` function. Any type that is an instance of the `Typeable` class can be represented as a `Dynamic` value.

Using the `Dynamic` type is generally considered poor functional programming practice, and frequently there are ways to re-express a solution to avoid its use. However, `Dynamic` is actually type-safe in the sense that run-time errors cannot occur because of the use of the "wrong" type. The reason using `Dynamic` is frowned upon somewhat is that the type system is not able to help the user with the process of his or her construction of CAL code as much. There is also a (small) performance penalty of carrying type information at runtime.

Data constructors

The `Dynamic` type has no public data constructors. `Dynamic` values can only be created using the `toDynamic` function.

Examples

```
listOfStringsAndNumbers :: [Dynamic];
listOfStringsAndNumbers = [toDynamic "fifty-five", toDynamic
55.0, toDynamic (55 :: Int)];

showStringOrDouble :: Dynamic -> String;
showStringOrDouble x =
    let
        doubleValue :: Maybe Double;
        doubleValue = fromDynamic x;
    in
        case doubleValue of
        Nothing -> fromDynamicWithDefault x "(not a String or
Double)";
        Just value -> Debug.show value;
        ;
```

With `showStringOrDouble` and `listOfStringsAndNumbers` defined as above:

```
map showStringOrDouble listOfStringsAndNumbers
returns ["fifty-five", "55.0", "(not a String or Double)"]
```

4.1.2 Either a b

The `Either` type represents values with two possibilities. A value of type `Either a b` is either `Left a` or `Right b`. For example, the list `[Left "abc", Right 2.0]` has type `Either String Double`.

The `Either` type is sometimes used as an alternative to the `Maybe` type when representing the return type of a function that may fail. The `Left` data constructor is then used to hold failure information (i.e. an error message for example), and the `Right` data constructor is used to hold the successful return value.

Data constructors

```
Left     value :: a
Right    value :: b
```

Examples

The `Either IOErrorType a` type is used by the `File` module to represent the result of functions that might signal an error.

```
/**
 * Determines whether an
 * {@code ({@link Either@} {@link typeConstructor=IOError@} a)@}
 * value represents an IO error, i.e. in fact the value is of the
 * form {@code {@link Left@} ioError@}. This function is
 * therefore analogous to {@link Prelude.isJust@}.
 *
 * @arg resultOrError an {@code (Either IOError a)@} value to be
 *                    processed.
 * @return {@link True@} if the IO operation succeeded, or
 *         {@link False@} if the IO operation failed.
 */
isIOSuccess :: Either IOErrorType a -> Boolean;
public isIOSuccess !resultOrError =
    case resultOrError of
    Left {} -> False;
    _ -> True;
    ;
```

4.1.3 Maybe a

The `Maybe` type can be thought of as representing an optional value. For example, a value of type `Maybe Double` can be `Just 2.0`, indicating that the value `2.0` was supplied, or it can be `Nothing`, indicating that no `Double` value was supplied.

Functions that perform operations that could fail (such as database access) frequently have a return type of `Maybe`, with a return value of `Nothing` indicating failure.

The `fromJust` function can be used to extract the wrapped value from a `Just` value. The `maybeToList` function will convert a `Maybe` value to either a single-element list of the wrapped value (in the `Just` case) or an empty list (in the `Nothing` case).

Data constructors

```
Nothing
Just    value :: a
```

Examples

```
/**
 * {@code lookup key map@} returns the value corresponding to key
 * in the map (association list).
 *
 * @arg key the key for which the map is to be searched.
 * @arg map the map (association list) to be searched.
 * @return the value corresponding to key in the map.
 */
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b;
public lookup key !map =
    case map of
    [] -> Nothing;
    mapHead : mapTail ->
        if key == mapHead.#1 then
            Just mapHead.#2
        else
            lookup key mapTail;
    ;
```

With lookup defined as above:

```
lookup "apple" [("orange", 2.49), ("apple", 1.29), ("pear",
3.29)]
returns Just 1.29
```

```
lookup "pomegranate" [("orange", 2.49), ("apple", 1.29), ("pear",
3.29)]
returns Nothing
```

4.1.4 Ordering

`Ordering` is a simple enumerated type intended to represent the result of a comparison between two values.

Data constructors

`LT`	Represents "less than"
`EQ`	Represents a comparison of equal values
`GT`	Represents "greater than"

Examples

```
map (compare (0 :: Int)) (upFromTo (-2) 2)
returns [LT, LT, EQ, GT, GT]
```

4.1.5 TypeRep

`TypeRep` values represent the type of expressions. The `TypeRep` type exists primarily to support dynamically-typed programming using the `Dynamic` type.

Data constructors

The `TypeRep` class has no public data constructors. The only way to create a `TypeRep` value is by applying the `typeOf` method to an expression.

Examples

```
typeOf "str1" == typeOf "str2"
returns True

typeOf EQ == typeOf LT
returns True

typeOf (4 + 5.0) == typeOf (4 + (5 :: Int))
returns False
```

4.2 Type classes

4.2.1 Eq

`Eq` is the class of types that can be compared for equality.

Methods

`equals`	Returns `True` if the two arguments are equal. The operator form of this method is `==`.
`notEquals`	Returns `True` if the two arguments are not equal. The operator form of this method is `!=`.

Examples

```
"str1" == "str2"
returns False

EQ != LT
returns True

equals 'c' 'c'
returns True
```


4.2.2 Ord

`Ord` is the class of types whose values have an order. Any type that is an instance of `Ord` can be used as input to the ordering operators (e.g., `False < True` is a valid expression because `Boolean` is an instance of `Ord`).

Methods

`lessThan`	Returns `True` if the first argument is less than the second. The operator form of this method is `<`.
`lessThanEquals`	Returns `True` if the first argument is less than or equal to the second. The operator form of this method is `<=`.
`greaterThan`	Returns `True` if the first argument is greater than the second. The operator form of this method is `>`.
`greaterThanEquals`	Returns `True` if the first argument is greater than or equal to the second. The operator form of this method is `>=`.
`compare`	Return `EQ` if the first argument is equal to the second, `LT` if the first argument is less than the second, and `GT` if the first argument is greater than the second.
`max`	Returns the greater of its two arguments
`min`	Returns the lesser of its two arguments

Examples

```
LT <= EQ
returns True
```

```
0 :: Int > 10
returns False
```

```
"this" > "that"
returns True
```

```
max "this" "that"
returns "this"
```

```
compare 0.0 1.0
returns LT
```

4.2.3 Num

`Num` is the class of numeric types (i.e., those that support the usual arithmetic operations). Any type that is an instance of `Num` can be used as input to arithmetic operators (e.g., `4.0 + 3.0` is a valid expression because `Double` is an instance of `Num`).

Methods

`fromInteger`	Converts an `Integer` to the nearest value of the instance type.
`toDouble`	Converts a value of the instance type to the nearest `Double` value.
`negate`	Returns a value of opposite sign. The operator form of this method is the unary `-`.
`abs`	Returns the absolute value of a numeric value.
`signum`	Returns a value representing the sign of a number: `-1` for negative numbers, `0` for `0`, and `1` for positive numbers.
`add`	Adds two numbers. The operator form of this method is `+`.
`subtract`	Subtracts two numbers. The operator form of this method is binary `-`.
`multiply`	Multiplies two numbers. The operator form of this method is `*`.
`divide`	Divides two numbers. The operator form of this method is `/`.
`remainder`	Returns the remainder from dividing two numbers. The operator form of this method is `%`.

Examples

```
negate 7 :: Int
returns -7

3 * 9.0
returns 27.0

signum (-900)
returns -1

fromInteger 500 :: Float
returns 500.0
```

4.2.4 Inputable

`Inputable` is the class of all types that can be converted from Java objects to appropriate native CAL values using the `input` method. This class and the `Outputable` class help to simplify integration between Java and CAL code by providing a mapping between native CAL values and their Java equivalents (for example, between native CAL lists and Java arrays or `List` objects).

Methods

`input`	Converts a `JObject` to an appropriate native CAL value.

4.2.5 Outputable

`Outputable` is the class of all types that can be converted from native CAL values to Java objects using the `output` method. This class and the `Inputable` class help to simplify integration between Java and CAL code by providing a mapping between native CAL values and their Java equivalents (for example, between native CAL lists and Java arrays or `List` objects).

Methods

`output`	Converts a native CAL value to an appropriate `JObject`.

Examples

4.2.6 Bounded

`Bounded` is the class of all types that have upper and lower bounds on their values.

Methods

`minBound`	Returns the smallest possible value of this type.
`maxBound`	Returns the largest possible value of this type.

Examples

```
minBound :: Byte
returns -128

maxBound :: Int
returns 2147483647

maxBound :: Ordering
returns GT
```

4.2.7 Appendable

`Appendable` is the class of all sequence types that can be joined together using `append` or `concat`.

Methods

`empty`	Returns the value that represents the empty sequence for this type.
`isEmpty`	Returns true if its argument is an empty sequence.
`append`	Returns a new value that consists of the concatenation of its two arguments. The operator form of this method is `++`.
`concat`	Returns a new value that consists of the concatenation of the elements of its argument list. This method is provided for efficiency reasons. `"a" ++ "b" ++ "c" ++ "d"` generates 2 intermediate values, whereas `concat ["a", "b", "c", "d"]` does not generate any intermediate values.

Examples

```
append [2, 1, 2] [4, 5, 6, 7, 8] :: [Int]
returns [2, 1, 2, 4, 5, 6, 7, 8]

concat ["string1", " ", "string 2", " ", "string 3"]
returns "string1 string2 string3"

concat [[12, 12], [6], [12, 12]]
returns [12, 12, 6, 12, 12]

empty :: String
returns ""

empty :: [Int]
returns []

isEmpty [1.0]
returns False

isEmpty ""
returns True
```

4.2.8 Typeable

The `Typeable` class is the class of all types that have an associated `TypeRep` representation. All non-polymorphic CAL types, and all polymorphic types whose type argument variables are not higher-kinded are automatically instances of the `Typeable` class. There is no need to explicitly define `Typeable` instances for CAL types.

The `Typeable` class exists primarily to support dynamically-typed programming using the `Dynamic` type (see Dynamic typing and the Dynamic type for more details).

Methods

`typeOf`	Returns a `TypeRep` value representing the type of a value.

Examples

```
(typeOf "str") == (typeOf 'c')
returns False

(typeOf "string1") == (typeOf "string2")
returns True
```

4.2.9 Enum

`Enum` is a type class intended to represent types whose values can be enumerated one by one, such as `Int`, `Long`, `Integer` and `Ordering`. It is also used for types such as `Double`, where an enumeration can be defined on a subset of the values, such as the series of values `1, 1.5, 2, 2.5, 3`.

`upFrom`	For numeric types, creates an ascending list starting from its argument.
`upFromThen`	For numeric types, creates a list starting with the two arguments, and then continuing by the difference of the two (see examples)
`upFromTo`	For numeric types, creates an ascending list starting from its first argument and continuing until it gets to the second argument.
`upFromThenTo`	For numeric types, creates a list starting with the two arguments, and then continuing by the difference of the two, and continuing until it gets to the third argument. (see examples)

Examples

```
zip ["item_id", "item_description", "order_id"] (upFrom 1 ::
[Int])
returns [("item_id", 1), ("item_description", 2), ("order_id",
3)]
```


4.2.10 IntEnum

`IntEnum` is a type class that represents types where there is a mapping between the values of the type and the values (or a subset of the values) of the `Int` type.

Enumerations (algebraic types whose data constructors all take 0 arguments) often need to be translated to and from `Int` values so that they can be stored in external locations (e.g., databases, files, preferences). This class makes it easy to generate the translation functions for such enumerations by deriving an instance.

For all values `x`, the following must be true:

```
intToEnum (enumToInt x) == x
```

It is possible for multiple `Int` values to map to the same value of the instance type. However, each instance type value must map only to a single `Int` value.

Instances of this type can be derived for algebraic data types that are enumeration types (i.e., non-polymorphic algebraic data types where each data constructor takes zero arguments).

Methods

`intToEnum`	Returns the value that corresponds to the specified `Int` value. Raises an error if there is no mapping for the provided `Int` value.
`intToEnumChecked`	Returns `Just value`, where `value` is the value that corresponds to the specified `Int` value. If there is no mapping for the provided Int value, returns `Nothing`.
`enumToInt`	Returns the `Int` value that corresponds to the provided value.

Examples

```
data public MyColorEnum =
    Red |
    Green |
    Blue |
    Other
    deriving Enum, IntEnum;
```

With the above definition of `MyColorEnum`:

```
enumToInt Red
returns 0
```

```
enumToInt Other
returns 3
```

```
intToEnum 2
returns Blue

intToEnum 500
Error: argument (100) does not correspond to a value of type
MyColorEnum
```

4.3 Functions and methods

This section presents a list of the (approximately) 50 most-commonly-used CAL functions and methods at the time that this document was written. A brief description is given for each function or method.

The functions reside in different modules, so their fully-qualified names are given (e.g., `Prelude.compare` instead of just `compare`).

List.all pred list	Returns `True` if the provided predicate evaluates to `True` on each element of the list.
Prelude.append a1 a2	(method) Appends two `Appendables` together. `Strings` and lists are both `Appendable`.
List.chop len list	Returns list chopped into sublists of length `len`. ex: `chop 1 [1.0, 2.0, 3.0]` is `[[1.0], [2.0], [3.0]]`
Prelude.compare o1 o2	(method) Compares two `Ords` and returns `LT`, `GT`, or `EQ`
Prelude.concat list	(method) Concatenates a list of `Appendables` into a single list
List.concatMap f list	Applies the function `f` to each element of list and then concatenates the resulting list.
Prelude.const k x	Always returns `k`
Prelude.doubleToString num	Returns the `String` representation of a `Double`
Prelude.equals e1 e2	(method) Returns `True` if the two provided `Eqs` are equal, and `False` otherwise.
Prelude.error str	Raises an error
List.filter pred list	Returns a list of each element of list for which pred returns `True`.
List.foldLeft f init list	"folds" the 2-argument function f among the elements of list in a left-associative fashion into a single result. Ex: `foldLeft add 0 [1,2,3] = (add (add (add 0 1) 2) 3)`
List.foldLeftStrict f init list	Strict version of `foldLeft`. Equivalent to `foldLeft`, except that evaluation is forced at each stage of the computation. This version is a used for efficiency reasons in certain situations.
List.foldRight f init list	"folds" the 2-argument function f among the elements of list in a right-associative fashion into a single result. Ex: `foldRight divide 1 [1,2,3] = (divide 1 (divide 2 (divide 3 1)))`
Prelude.fromJust m	Converts a value `m = Just x` into `x`.
Prelude.fst tuple	Returns the first component of a 2-tuple (i.e. pair)

List.head list	Returns the first element of a list
Prelude.input obj	(method) Converts a Java object into an appropriately-typed CAL object
List.intersperse x list	Returns a list with x between each two elements of list. Ex: `intersperse 0 [1,2,3] = [1,0,2,0,3]`
Prelude.intToString int	Returns the String representation of an Int
Prelude.isNothing m	Returns True if m is Nothing, and False if m is Just x for some x.
List.last list	Returns the last element of list
List.length list	Returns the length of list
List.list2 x y	Returns a 2-element list of x and y
List.map f list	Returns a list of the results of f applied to each element of list. Ex: `map doubleToString [1.0, 2.0] = ["1.0", "2.0"]`
Prelude.max a b	Returns the greater of a and b
Prelude.multiply num1 num2	(method) x multiplied by y
Prelude.not bool	Logical NOT of a boolean value
Prelude.output val	(method) Converts a CAL values into an appropriately-type Java object
Prelude.outputList list	Converts a CAL list of CAL values into a Java list of appropriately-typed Java objects
Prelude.remainder x y	(method) The remainder of x divided by y.
List.reverse list	Returns a list whose elements are the reversed elements of list. Ex: `reverse [1,2,3] = [3,2,1]`
Prelude.round num	Round a number to the nearest integer
Prelude.seq a b	Force the order of evaluation. Seq evaluates a until it is in WHNF (weak head normal form), and then returns b
Debug.show x	(method) Returns the String representation of x
Prelude.sin x	Trigonometric sine function
Prelude.snd tuple	Returns the second component of a 2-tuple (i.e. pair)
String.toList str	Returns a list of the `Chars` that a string contains. Ex: `stringToCharacters "hello" = ['h','e','l','l','o']`
List.subscript list n	Returns the nth element of list. N is zero-based. Ex: `subscript 2 [1,2,3,4] = 3`
List.sum list	Returns the sum of a list of numbers.
List.tail list	Returns a list minus its first element. Ex: `tail [1,2,3] = [2,3]`
List.take n list	Returns the first n elements of list
Prelude.toDouble num	Converts a Number to a Double
Prelude.field1 tuple	Returns the first component of a tuple (or the `#1` field of an arbitrary record)
Prelude.field2 tuple	Returns the second component of a tuple (or the `#2` field of an arbitrary record)

Prelude.field3 tuple	Returns the third component of a tuple (or the `#3` field of an arbitrary record)
Prelude.upFrom n	Returns an infinite list of numbers starting from n and increasing by one. Ex: `upFrom 5.0 = [5.0, 6.0, 7.0, ...]`
Prelude.upFromTo start end	Returns a list of numbers starting from start and ending with end, inclusive. Ex: `upFromTo 5 10 = [5,6,7,8,9,10]`
List.zip list1 list2	Returns a list of pairs, where the first element of the nth pair is the nth element of list1, and the second element of the nth pair is the nth element of list2. Ex: `zip [1,2,3] ['a','b','c'] = [(1,'a'), (2,'b'), (3,'c')]`
List.zip3 list1 list2 list3	Returns a list of 3-tuples, where the first element of the nth pair is the nth element of list1, the second element of the nth pair is the nth element of list2, and the third element of the nth triple is the nth element of list3. Ex: `zip3 [1,2,3] ['a','b','c'] [5,6,7]= [(1,'a',5), (2,'b',6), (3,'c',7)]`
List.zipWith f list1 list2	Returns a list where the nth element is the result of applying the 2-argument function f to the nth element of list1 and the nth element of list 2. Ex: `zipWith add [1,2,3] [4,5,6] = [5,7,9]`

5 Appendices

5.1 CAL source formatting conventions

There are a number of conventions for formatting CAL source code. This section lists these conventions with examples. The general principle informing all of the formatting conventions is "indent as little as possible consistent with showing the logical structure and scoping of the program".

5.1.1 General guidelines

- Use 4 spaces for each level of indentation.
- Don't use tabs.
- Insert a newline after an expression defining equals, and indent the expression one level of scope, unless the whole defining expression fits easily on one line.
- Insert newlines and start a new scope after `=`, `then`, `else`, `->`, `let`, `in`, and `where`. (but not after `of`. See the Formatting case statements below for detail.)
- It is good practice to include explicit type declarations even if they can be inferred.

5.1.2 Formatting if-then-else expressions

Here are some concrete examples of correctly-formatted `if` statements.

```
toEnumForOrdering :: Int -> Ordering;
private toEnumForOrdering index =
    if index == 0 then
        LT
    else if index == 1 then
        EQ
    else if index == 2 then
        GT
    else
        error "Prelude.toEnumForOrdering: the index is out of
range."
    ;

maxOrd :: Ord a => a -> a -> a;
public maxOrd x y = if x >= y then x else y;
```

The first form (`toEnumForOrdering`) should be considered the default, and is the preferred form when the `then` and `else` expressions are long.

5.1.3 Formatting case expressions

Here are some concrete examples of correctly-formatted `case` expressions.

```
equalsOrdering :: Ordering -> Ordering -> Boolean;
private equalsOrdering !x !y =
    case x of
```



```
        LT ->
            case y of
            LT -> True;
            _ -> False;
            ;
        EQ ->
            case y of
            EQ -> True;
            _ -> False;
            ;
        GT ->
            case y of
            GT -> True;
            _ -> False;
;

 catMaybes :: [Maybe a] -> [a];
public catMaybes ms =
    case ms of
    [] -> [];
    x : xs ->
        case x of
        Nothing -> catMaybes xs;
        Just b  -> b : catMaybes xs;
        ;
    ;
```

The most important point in formatting cases is to make sure that the `case` keyword aligns with its corresponding alternative patterns and its expression terminating semicolon (if needed). This greatly improves the readability of nested `cases`, as in the `catMaybes` example.

It is very tempting to slightly abuse the minimal whitespace principle by aligning the arrows to improve clarity when the expressions following the pattern are short and thus kept on the same line as the arrows. Ex: the arrows following `Nothing` and `Just b` have been aligned vertically by inserting an extra space after `Just b`. Although this isn't strictly correct, it is not frowned upon.

5.1.4 Formatting let expressions

```
lines :: [Char] -> [[Char]];
public lines s =
    if isNull s then []
    else
        let
            lineRestPair :: ([Char], [Char]);
            lineRestPair = break (\c -> '\n' == c) s;

            line :: [Char];
            line = fst lineRestPair;

            rest :: [Char];
            rest = snd lineRestPair;
        in
            line : (if isNull rest then [] else lines (tail
rest));
```



```
maximumBy :: (a -> a -> Ordering) -> [a] -> a;
public maximumBy comparisonFunction list =
    case list of
    [] -> error "Prelude.maximumBy: empty list.";
    _ : _ ->
        let
            max x y =
                case (comparisonFunction x y) of
                GT -> x;
                _  -> y;
                ;
        in
            List.foldLeft1Strict max list;
    ;
```

Note that the `in` keyword lines up with its corresponding `let`, and that the expressions after both `let` and `in` are indented.

5.1.5 Formatting data declarations, type classes, and instances

The guidelines for formatting data declarations, type classes, and instances can mostly be inferred from the "least indentation principle". Some concrete examples:

```
data public Attribute =
    private ColourAttribute
        attrName      :: String
        colours       :: [Color] |
    private BooleanAttribute
        attrName      :: String
        boolVals      :: [Boolean] |
    private IntegerAttribute
        attrName      :: String
        intVals       :: [Int] |
    private DoubleAttribute
        attrName      :: String
        doubleVals    :: [Double] |
    private StringAttribute
        attrName      :: String
        stringVals    :: [String] |
    private TimeAttribute
        attrName      :: String
        timeVals      :: [Time] |
    private AttributeSetAttribute
        attrName      :: String
        childAttrSets :: [AttributeSet];

public class NamedDataFacet a where
    public getDisplayName :: a -> String;
    public getUniqueIdentifier :: a -> UniqueIdentifier a;
    ;

instance NamedDataFacet Field where
    getDisplayName = getFieldDisplayName;
    getUniqueIdentifier = getFieldUniqueIdentifier;
    ;
```


5.1.6 CALDoc style guidelines

Using the `@see` tag

When using the `@see` tag, it is recommended that references to type constructors, type classes and modules always appear with the appropriate context keywords. In other words, instead of writing:

```
/** @see Ord */
```

one should write:

```
/** @see typeClass = Ord */
```

Also, if one @see block in a comment uses a context keyword, then all @see blocks in the comment should do so as well. For example:

```
/**
 * @see typeClass = Ord
 * @see function = compare
 */
```

Using the `{@code}` and `{@link}` inline tags

In a CALDoc comment, all CAL fragments should appear in either a `{@code}` or a `{@link}` inline tag. For example (taken from List.map):

```
/**
 * {@code map mapFunction list@} applies the function {@code mapFunction@}
 * to each element of the list and returns the resulting list.
 *
 * @arg mapFunction a function to be applied to each element of the list.
 * @arg list the list.
 * @return the list obtained by applying {@code mapFunction@} to
 *         each element of the list.
 */
```

Moreover, names of top-level functions, class methods, type and data constructors, and type classes should be hyperlinked with the `{@link}` tag. For example:

```
/**
 * This is a cross reference to Prelude.Right: {@link Right@}
 *
 * This is how you would write "Maybe a": {@code {@link Maybe@} a@}.
 *
 * This is a nice example fragment:
 * {@code {@link List.map@} {@link truncate@} [2.6, -2.7] == [2, 2]@}
 *
 * To disambiguate, you would write {@link typeConstructor = String@}
 * or {@link module = String@}.
 */
```

Using the `@author` tag

It is recommended to put only one author name per `@author` block, and let the documentation generator handle the generation of commas. For example:


```
/**
 * @author Bo Ilic
 * @author Joseph Wong
 */
```

5.2 Suggested reading

CAL is influenced by the Haskell language, so some knowledge of Haskell is an advantage to learning CAL. These are two excellent introductions to Haskell (and to functional programming in general):

- *The Craft of Functional Programming* by Simon Thompson
- *A Gentle Introduction to Haskell* (available at http://www.haskell.org/tutorial/)

5.3 Language keywords

- `case`
- `class`
- `data`
- `dataConstructor`
- `default`
- `deriving`
- `else`
- `foreign`
- `friend`
- `function`
- `if`
- `import`
- `in`
- `instance`
- `jvm`
- `let`
- `module`
- `of`
- `primitive`
- `private`
- `protected`
- `public`
- `then`
- `typeClass`
- `typeConstructor`
- `unsafe`
- `using`
- `where`

