A First Course on Aldor with libaldor

Peter Broadbery Manuel Bronstein

March 8, 2002

1 Introduction

The Aldor language (formerly known as A$^\sharp$ and axiomxl) is the language used by the extension compiler in the Axiom system. It also allows you to create stand-alone executables which may be distributed without Axiom.

These notes are not intended to be a complete description of the language, but show you how to get started with Aldor and how to use the compiler. A more complete description is contained in the user guide.

You must have libaldor properly installed and compiled before running this tutorial, as well as have your environment variables ALDORLIBROOT, INCPATH and LIBPATH properly set. See the libaldor user guide for more information.

2 First Contact

Although Aldor is primarily a compiled language, the use of an interactive interpreted system is often praised as an aid for the rapid prototyping of programs. To address this need for rapid feedback, the compiler provides an interactive interpreter for evaluating expressions in the Aldor language.

These notes contain examples of interactive input to the Aldor interpreter. There are three kinds of lines in the text that follows:

- Lines starting with “>>” represent commands that you can type into the interpreter (you should not type the “>>” when you type the command).
- Lines immediately following the input represent the output from the interpreter.

- Lines starting with "`--`" represent comments in `Aldor` and serve to explain the following input commands.
- The interpreter prefixes every prompt with an index number

It is possible to retrieve previously computed values via the command:

```
#int history on
```

To begin an interactive session, start by typing "`aldor -gloop`" at the command line. The interpreter will respond with a prompt which includes a line number and a prompt: ">>". You can now enter commands at the prompt. Sections 2.1 to 2.6 are meant to be ran in the same interactive session. If you prefer copying and pasting rather than typing commands, all the input lines for the interactive sessions of this tutorial are provided in the files `session1.as`, `session2.as`, `session3.as` and `session4.as`.

2.1 Values

`Aldor` doesn't have any built-in knowledge about the values (for example numbers, strings and identifiers) it is likely to see, and these have to be loaded to get a reasonable environment (this is the same as "`#include` " in C).

```
-- basic include for all aldorlib clients
>> #include "aldor"

-- include the following only when using an interactive session
>> #include "aldorinterp"
```

We can now look at certain values

```
>> "hello"
hello @ String
```

In the `Aldor` world, integer literals (for example the character sequence "123") may represent different types (for example 32-bit machine integers, arbitrary precision integers, integers mod n, etc), so the default is that *no* integer type is in scope.

```
>> 1
    ^
[L4 C1] #1 (Error) No meaning for identifier ‘1’.
```

To put a type into scope, it must be imported. Importing arbitrary precision integers isn't a problem:

```
>> import from Integer
```

`Integer` here is a type which exports lots of useful operations on arbitrary-precision integers. We discuss domains in more depth later.

Now we can do some calculations:

```
>> 1
1 @ AldorInteger
>> 123456789*98765432
12193263098917848 @ AldorInteger

-- The normal precedence rules apply for infix operators:
>> 4 + 3 * 2
10 @ AldorInteger

-- Parentheses are used for grouping
>> 2*(3+4)
14 @ AldorInteger

-- Function calls can be written with parentheses around the arguments
>> next(10)
11 @ AldorInteger

-- Where there is a single argument, the parentheses are optional, and
-- aguments associate to the right.
>> next next 10
12 @ AldorInteger

-- Application has higher precedence than all arithmetic operators
>> next 2 * next 3
12 @ AldorInteger
```

There are several other numeric types in the `libaldor` library. These include:

`MachineInteger` machine-precision integers

`MachineFloat` single precision (32-bit) floats

`Boolean` boolean values (true/false)

2.2 Control

`Aldor` supports a variety of control and conditional constructs.

```
-- Blocks are sequences of expressions enclosed in braces.  Normally
-- each expression is evaluated in order with the last value returned.
>> {2; 3; 4; 5}
5 @ AldorInteger

-- A "fat arrow" expression can cause an early abnormal exit.
>> {2 < 3 => 4; 5}
4 @ AldorInteger

-- The expression "a => b" means "if a then exit (with value) b".
>> (2 < 1 => 4; 5)
5 @ AldorInteger

-- You can also write the previous two expressions as "conditionals".
>> if 2 < 3 then 4 else 5
4 @ AldorInteger

>> if 2 < 1 then 4 else 5
5 @ AldorInteger

-- You can store values into a local variable using infix :=.
>> x := 2 + 2
4 @ AldorInteger

-- The value of the local variable can be used in other expressions.
>> y := {2 < x => 7; 11}
7 @ AldorInteger

-- There are practically no restrictions as to what kinds of
-- expressions can be nested within others.
>> (y := 2) + (2 < 3 => 7; 11) * (if 2 < 1 then (w := 4) else 5)
37 @ AldorInteger

-- Printing is done using "stdout" with the << operator.
>> stdout << "The value of x + y is " << x + y
The value of x + y is 6  () @ TextWriter

-- Printing "newline" causes printing to begin on the next line.
>> stdout << "The value is " << (if x < y then 2+2 else 2+3) << newline
The value is 5
  () @ TextWriter
```

Note that normal `Aldor` syntax requires semi-colon or a closing bracket to terminate a statement. However, the interpreter is more liberal and will treat a newline as a statement terminator.

You may have noticed that the type of a print call is "`TextWriter`". This is a very flexible output type used by the library as a standard way of printing information. It can also be used for output to strings, filters, etc.

Note that the value returned by an output statement is printed as "()" — this indicates that the interpreter didn't find a method for printing the value.

2.3 Function definitions

```
-- Functions are created using infix +->.
-- You must supply the parameter and target types.
>> increment := (x: Integer): Integer +-> x + 1
  () @ (x: AldorInteger) -> AldorInteger

-- A function can be stored in a local variable like any other value.
>> increment 2
3 @ AldorInteger

-- The arguments supplied to a function must have the correct type.
>> increment "hello"
   ..........^
[L25 C11] #1 (Error) Argument 1 of `increment' did not match with
 any possible parameter type.
    The rejected type is String.
    Expected type AldorInteger.

-- The usual way to define a function is using an ==.
>> inc(x: Integer): Integer == x + 1
Defined inc @ (x: AldorInteger) -> AldorInteger

>> inc 2
3 @ AldorInteger

-- The == signifies that "inc" is a constant (and should not be changed).
-- Reply 'n' when asked whether to redefine inc.
>> inc(x: Integer): Integer == x + 2
Redefine? (y/n): n
   ^
[L28 C1] #1 (Error) Constant `inc' cannot be redefined.

-- Remember that function application has higher precedence
-- than arithmetic operations.
>> inc 2*3
9 @ AldorInteger

-- Define the factorial function.
>> fact(n: Integer): Integer == {
        n < 2 => 1;
        n * fact (n - 1)
   }
```

```
Defined fact @ (n: AldorInteger) -> AldorInteger

>> fact 30
265252859812191058636308480000000 @ AldorInteger
```

Aldor also allows default values for arguments:

```
>> incr(x: Integer, amount: Integer == 1): Integer == x + amount;
Defined incr @ (x: AldorInteger, amount: AldorInteger == 1) -> AldorInteger
>> incr(12)
13 @ AldorInteger

>> incr(13,7)
20 @ AldorInteger
```

Here incr is defined to take two arguments, but the second may be omitted when calling the function.

2.4 Collections

```
-- Now for some fun with lists.
>> import from List Integer

-- The bracket operation ('[' and ']') creates a list from its
-- elements.  The operation is imported from List, not a builtin
-- function.
>> [1, 3, 5, 7, 9, 11]
[1,3,5,7,9,11] @ List(AldorInteger)

-- Another way to create the same list is to use a loop.
>> [i for i in 1..11 by 2]
[1,3,5,7,9,11] @ List(AldorInteger)

-- Loops can count backwards as well.
>> [2*i for i in 11..1 by -2]
[22,18,14,10,6,2] @ List(AldorInteger)

-- Another way is to use a "such that" clause (introduced by '|').
>> [2*i for i in 11..1 by -1 | odd? i]
[22,18,14,10,6,2] @ List(AldorInteger)

-- Let us bring machine integers and lists of them in scope too,
-- this will create some ambiguity for integer constants!
>> import from MachineInteger, List MachineInteger;
```

```
-- Since both MachineInteger and Integer are now in scope
-- the expression -1 could mean two different things.
>> k := -1
   ^
[L66 C1] #1 (Error) The type of the assignment cannot be inferred.
     The possible types of the right hand side (‘- 1’) are:
          -- MachineInteger
          -- AldorInteger

-- You can specify which value you mean using a declaration.
>> k: MachineInteger := -1
-1 @ MachineInteger

-- Another way to specify is to restrict an expression (using ’@’)
-- to a particular type.
>> m := (-1 @ Integer)
-1 @ AldorInteger

-- A while construct tests before a value is collected (ignore the warning)
>> u := [(k := k + 2) while k < 11]
[1,3,5,7,9,11] @ List(MachineInteger)
       .......^
[L69 C8] #1 (Warning) Implicit local ‘k’ is a parameter, ...


-- You can also iterate over a list-valued expression.
>> v := [z for z in u | z < 9]
[1,3,5,7] @ List(MachineInteger)

-- To extract an element of a list, apply the list to an index.
>> ww := [u.i for i in 1..5@MachineInteger by 2]
[1,5,9] @ List(MachineInteger)

-- the "@MachineInteger" indicates that the integer 5 should be
-- treated as a MachineInteger.
-- This can be avoided with declaring i to have a default type
>> default i: MachineInteger
 @ Category ==

-- Two for-constructors can be given in parallel.
>> [i*j for i in 1..10 for j in u]
[1,6,15,28,45,66] @ List(MachineInteger)

-- In general, any number of constructs can be given in parallel.
-- The following runs over the 1, 3, 5, 7, 9 in parallel with
-- the values in u, collecting the products p of the
-- corresponding elements until p exceeds 24.
>> [p for i in 1..10 | odd? i for j in u while (p := i*j) < 24]
[1,9] @ List(MachineInteger)
```

```
-- Constructs may be arbitrarily nested but you must import
-- what you need.
>> import from List List MachineInteger

-- Now we can form lists of lists of integers.
>> [[i*j for i in 1..2] for j in u]
[[1,2],[3,6],[5,10],[7,14],[9,18],[11,22]] @ List(List(MachineInteger))
```

2.5 Iterations

```
-- Display the even-numbered integers between -5 and +5
>> for i in -5..5 | even? i repeat stdout << i << space
-4 -2 0 2 4

-- To go on to the next iteration, use "iterate"
>> for i in -5..5 repeat {
        odd? i => iterate;
        stdout << i << space
   }
-4 -2 0 2 4

-- To exit a loop early, use "break"
>> for i in -5.. repeat {
        odd? i => iterate;
        i > 5 => break;
        stdout << i << space
   }
-4 -2 0 2 4

-- As with collections, iterators can use while
>> { x:= 0;
     while x < 10 repeat { stdout << x << " "; x := x+1 }
     stdout << newline }
0 1 2 3 4 5 6 7 8 9
  () @ TextWriter

-- without any guards, repeat will iterate forever
>> repeat { stdout << "Hit Ctrl-C to stop " << newline }
Hit Ctrl-C to stop
Hit Ctrl-C to stop
Hit Ctrl-C to stop
...
```

2.6 General iteration

Aldor provides a builtin type Generator which allows one to treat iteration in a uniform way. This allows the "`f(x) for x in ...`" examples above to be applicable to many different types.

For example,

```
>> l: List MachineInteger := [i for i in 1..10];
[1,2,3,4,5,6,7,8,9,10] @ List(MachineInteger)

>> for i in l repeat { stdout << i << space }
1 2 3 4 5 6 7 8 9 10

>> import from Array MachineInteger;

>> a: Array MachineInteger := [i for i in 1..10];
[1,2,3,4,5,6,7,8,9,10] @ Array(MachineInteger)

>> for i in a repeat { stdout << i << space }
1 2 3 4 5 6 7 8 9 10
```

In `Aldor`, the `Array` version looks identical to the `List` version, but the algorithms for building and traversing the two objects are quite different. In fact, `Aldor` makes it easy to write such iteration routines through a type called `Generator`.

The "`generate`" construct is used to build a generator. For example, say we wanted a way of looping over two user supplied lists (of integers, for example), alternately selecting from one then the other. That is we want to be able to say something like:

```
for x in both(a, b) repeat g(x)
```

and have `g` called with the first item in `a`, then the first in `b`, then the second in `a`, and so on. Obviously we could achieve the same thing by merging the lists, and then iterating over the combined list, but this wastes storage if we don't modify the lists in-place, and time and effort if we do.

Providing that we know what `g` is, we can write this pretty easily[1]:

```
both(l1: List Integer, l2: List Integer): () == {
        for s1 in l1 for s2 in l2 repeat {
                g(s1);
                g(s2);
        }
}
```

[1] do not type this function in your interactive session, since the function `g` is undefined

Did I mention the lists were the same length? In any case, if we don't know what `g` is, or we are trying to "do something" with the value returned by `g`, we need to go back and change this code. If the iteration were more complex (for example, we had a tree and a list rather than two lists), this will start to become unpleasant.

So, we try to abstract what this iteration is doing, and try to turn that into a real object. How about:

```
>> both(l1:List Integer, l2:List Integer):Generator Integer == {
        generate {
                for s1 in l1 for s2 in l2 repeat {
                        yield s1;
                        yield s2;
                }
        }
}
Defined both @
    (l1: List(AldorInteger), l2: List(AldorInteger)) -> Generator(AldorInteger)
```

Here the word "`generate`" wraps the body and "`yield`" is used where we want to produce a value.

The "`generate`" form constructs an "object" that when asked for a new value will execute its body in the usual way until a "`yield`" statement is reached. The yield then evaluates its argument, and this value is then given back to the requester of the value. When another value is requested, control starts immediately after the yield and proceeds up to the next yield. The "request" for a value is such that the generator object can say if no more values are available, so this process will finish nicely.

Once we have have a Generator in our hands, we can iterate over it (this is with the interpreter):

```
-- Something simple:
>> for z in both([1,2,3,4,5],[5,4,3,2,1]) repeat { stdout << z << space }
1 5 2 4 3 3 4 2 5 1

-- we can build the combined list using a collection:
>> [s for s in both([1, 2], [10, 3])]
[1, 10, 2, 3] @ List(AldorInteger)
```

Of course, one can write functions that work on generators producing a new one. For example:

```
>> filter(f: MachineInteger -> Boolean,
```

```
          g: Generator MachineInteger): Generator MachineInteger == {
              generate {
                  for z in g repeat { if f(z) then yield z }
              }
}
Defined filter @ (f: MachineInteger -> Boolean,
   g: Generator(MachineInteger)) -> Generator(MachineInteger)
```

This function can be used to select the interesting values from an iteration:

```
>> for z in filter(odd?, i*i for i in -10..10) repeat {
       stdout << z << space }
81 49 25 9 1 1 9 25 49 81
```

In this example, the inner "for" loop creates a generator which will generate the squares of all the integers from -10 to 10. This is then passed to the filter function, which produces a generator that selects only the odd ones.

It is important to note that no values are produced by the first line in the example. This allows us to apply "filter" to generators that produce an unbounded number of values.

```
-- Produces low, low+1,...,high - 1, high,
--          high-1, ..., low + 1, low, low + 1, etc...
>> upAndDown(low: MachineInteger,
             hi: MachineInteger): Generator MachineInteger == generate {
                 repeat {
                         for z in low..hi repeat yield z;
                         for z in (hi-1)..(low+1) by -1 repeat yield z;
                 }
}
Defined upAndDown @ (low: MachineInteger, hi: MachineInteger)
   -> Generator(MachineInteger)

>> for i in 1..10 for z in filter(even?, upAndDown(1,10)) repeat {
        stdout << z << space }
2 4 6 8 10 8 6 4 2 2
```

Note that the collection syntax introduced earlier for creating lists is really just a shorthand method for getting a generator, followed by a call to an operation "bracket", which constructs the list from the generator. In this way, the syntax for creating and manipulating sequences is quite uniform.

The Aldor compiler contains an optimizer which (amongst other things) can turn a generator construct into extremely efficient code. If the optimizer is not used, then iteration code does tend to be slow.

2.7 Streams

As we have seen, generators can be used to generate infinitely many elements. However, the collections we have seen until now are limited to containing only finitely many elements. `libaldor` provides a type `Stream` whose elements are infinite collections. There are many different ways to create a stream, the easiest being bracketing an infinite iterator:

```
-- Declaring a variable to be of a type T automatically imports from T
>> evens:Stream Integer := [n for n in 1@Integer .. | even? n]
[...] @ Stream(AldorInteger)
```

The result is displayed as [...] because streams are lazy object: their elements are computed only when requested by the user or by a computation involving the stream. Since no element of `evens` has been computed, no values are shown. However, streams store all the values previously computed in a dense array, so values are never computed more than once. You can thus think of streams as infinite generators with a memory.

```
-- Let's check the 6th positive even integer
>> evens.5
12 @ AldorInteger

-- See all the elements of evens that have been computed so far
>> evens
[2,4,6,8,10,12,...] @ Stream(AldorInteger)
```

There are many useful ways besides generators to create streams, for example by giving a function that computes the n^{th} element, or even a function that uses the elements $s.0, \ldots, s_{n-1}$ to compute the element $s.n$. Since values are stored once computed and never recomputed, streams provide an efficient way to evaluate recursive functions. For example, the Fibonacci numbers can be computed in linear time (and space) via a stream:

```
-- Given that s is a stream storing the Fibonacci numbers up to index n-1,
-- this function computes the Fibonacci number of index n
>> genfib(n:MachineInteger, s:Stream Integer):Integer == {
        n = 0 or n = 1 => 1;
        s(n-1) + s(n-2);
}
Defined genfib @ (n: MachineInteger, s: Stream(AldorInteger)) -> AldorInteger

-- This creates the stream of all the Fibonacci numbers
>> fib := stream genfib;
[...] @ Stream(AldorInteger)
```

```
-- Let's check the 15-th Fibonnaci number
>> fib 14
610 @ AldorInteger

-- See all the elements of fib that have been computed so far
>> fib
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,...] @ Stream(AldorInteger)

-- Streams can be iterated like any other data structure
-- but make sure to add another clause to avoid infinite loops!
>> for n in fib while n < 100 repeat stdout << n << space
1 1 2 3 5 8 13 21 34 55 89
```

3 Programs

The interpreter is a convenient way of testing programs and writing one-line functions, but for larger programs, it is easier to use the compiler.

The following file "`sort0.as`" is a short method for sorting an array in-place. Note that the "`!`" of "`bubbleSort!`" is part of the name of the function. It has no special meaning in `Aldor` but is used by convention in `libaldor` to indicate that this function modifies its argument.

```
#include "aldor"

bubbleSort!(arr: Array MachineInteger): Array MachineInteger == {
    import from MachineInteger;
    for i in #(arr)-1 .. 0 by -1 repeat {
        for j in 1..i repeat {
            if arr.(j-1) > arr.j then {
                t := arr(j-1);
                arr.(j-1) := arr.j;
                arr.j := t;
            }
        }
    }
    arr
}
```

To compile the file, do:

```
% aldor sort0.as
```

This will produce an intermediate file, `sort0.ao`. This contains a compiled version of the program, plus information on the names defined by `sort0.as`.

Once it is compiled, you can use it within the interpreter:

```
% aldor -gloop

>> #include "aldor"

>> #include "aldorinterp"

-- This tells the compiler that the file sort0.ao is interesting
>> #library Sort0 "sort0.ao"

-- Libraries are imported in the same way as domains:
>> import from Sort0

-- bubbleSort! should now be in scope.
>> bubbleSort!
() @ (arr: Array(MachineInteger)) -> Array(MachineInteger)

-- Now we try something:
>> bubbleSort! [4,1]
   .......^.^
[L6 C8] #1 (Error) No meaning for integer-style literal '4'.
[L6 C10] #2 (Error) No meaning for identifier '1'.

-- whoops, we haven't imported from MachineInteger yet.
>> import from MachineInteger

>> bubbleSort!([4,3,1,5,7])
%8 >> bubbleSort!([4,3,1,5,7])
      .......^
[L8 C8] #1 (Error) Argument 1 of 'bracket' did not match any possible...
    The rejected type is MachineInteger.
    Expected type Character.

-- Or arrays...
>> import from Array MachineInteger

-- This ought to work...
>> bubbleSort!([4,3,1,5,7])
[1,3,4,5,7] @ Array(MachineInteger)

-- How about a big array:
>> bubbleSort!([x for x in 100..1 by -1])
[1,2,3,4,5,6,7,8,9,10,...

-- The built-in sort! from Array MachineInteger should be somewhat faster:
>> sort!([x for x in 100..1 by -1])
[1,2,3,4,5,6,7,8,9,10,...
```

As it stands, `sort0.as` contains only a library function. We can turn it into a real program by adding asome top-level statements. `sort1.as` is the same as `sort0.as` with the following additions:

```
-- No need to call this "main", but an additional function
-- keeps the top-level name-space free from clutter.
main():() == {
    import from MachineInteger, Array MachineInteger;
    import from TextWriter;         -- for 'stdout'
    import from WriterManipulator; -- for 'endnl'
    arr := [x*(x-5) for x in 100..1 by -1];
    bubbleSort! arr;
    -- endnl sends a newline and flushes the stream
    stdout << arr << endnl;
}

-- call the function.
main();
```

Compile this with "`aldor -ginterp sort1.as`". This should compile the file, and execute the resulting intermediate code, printing the sorted array.

Of course, we could put this function into a new file and put the following at the head of the file:

```
#library mysort "sort0.ao"
import from mysort;
inline from mysort;        -- optional
```

to use the original library. The "`inline`" statement is optional, but will enable more optimisations to occur. It has a syntax similar to "`import`", but it states that the result of compilation is allowed to depend on the code generated for the libraries listed (not simply their existence). This allows the production of significanty better code, at the expense of having to recompile the file whenever the named libraries change.

To produce an executable, compile with

```
% aldor -fx -laldor -y$ALDORLIBROOT/lib sort1.as
```

This should then produce a file "`sort1`" which when run, prints the same result. It is also possible to use "`-grun`" which compiles the program (as with `-fx`), and then runs it, but does not save the executable for future runs. Typically `-ginterp` is a little faster than `-grun` as only one compilation stage is needed, but this depends on the execution time of the resulting program. In any case, if it is meant to be run more than once, then `-fx` is the preferred option.

The compiler has an optimizer which can improve performance dramatically. The "`-q3`" option to the compiler will invoke the optimizer before the `.ao` file is produced.

Use the "`-h`" option to the compiler to see what other options are available.

3.1 Some Language Features

In order to make Aldor programs more readable, a variety of shorthands are provided. We have alreay seen one of these — the `bracket` notation. In this case, an expression such as `[a]` is interpreted as `bracket(a)`.

Other examples are:

- `::` – "`a::T`" is equivalent to the form "`coerce(a)@T`".
- `apply` – "`arr.b`" is equivalent to the form "`apply(arr, b)`".
- `set!` – "`arr.b := c`" is equivalent to the form "`set!(arr, b, c)`".

4 Some useful types

In this section we describe some useful built-in types. They are commonly used as part of the representation of other objects. You should start a fresh interactive session for this section and include both `aldor` and `aldorinterp`.

4.1 Records

`Aldor` provides a basic structure type, called Record, which can be used to aggregate data, similar to `struct` in C.

A Record type looks like:

$$\texttt{Record(}id_1 \; \texttt{:} \;\; Type_1\texttt{,}\; id_2 \; \texttt{:} \;\; Type_2\texttt{, ...)}$$

The easiest way to explain is by example:

```
-- A record is a type, just like any other.
>> import from MachineInteger, Record(name: String, age: MachineInteger);
```

```
-- square brackets create a new object
>> newBod := ["Ethel T. Aardvark", 21];
 () @ Record(name: String, age: MachineInteger)

-- record elements are referenced by function application.
-- normally, a dot is used to ensure that the call parses as expected.
>> newBod.name
Ethel T. Aardvark @ String

>> newBod.age
21 @ MachineInteger

>> newBod(age)
21 @ MachineInteger

>> newBod age
21 @ MachineInteger

-- Records can be updated too:
-- Here the left side of the := is a function application.
>> newBod.age := newBod.age + 1;
22 @ MachineInteger
```

The function application notation for accessing and updating elements of a record are shorthands for the equivalent:

```
>> apply(newBod, age)
22 @ MachineInteger

>> set!(newBod, age, 21)
21 @ MachineInteger
```

Records are created through the bracket operation "`[]`" above, and also by a function called `record` for the same purpose (this is useful when there are many different bracket operations in scope).

4.2 Unions

`Aldor` provides a union type, called Union which is used to hold one of a selection of different types. This is similar to the "union" construct in C.

A Union type looks like:

$$\texttt{Union(}id_1\ \texttt{:}\ \ Type_1\texttt{,}\ id_2\ \texttt{:}\ \ Type_2\texttt{,}\ \texttt{...)}$$

This can contain an object of type $Type_1$, or an object of type $Type_2$, etc. The interface to this type is similar to that of records.

For example,

```
>> import from Union(s: String, i: MachineInteger);

-- create one of these chaps
>> u := union(12);
() @ Union(s: String, i: MachineInteger)

-- or you can use brackets
>> v := ["Hello"]
() @ Union(s: String, i: MachineInteger)

-- we can test which branch an object is a member of:
>> if u case s then stdout << "I'm a string with value: " << u.s;

-- we can destructively update the contents of a union
>> u.s := "A string";
A string @ String

-- and check what it is again:
>> if u case s then stdout << "I'm a string with value: " << u.s;
I'm a string with value: A string
```

4.3 Enumerations

Frequently, one needs a type whose elements come from some small set of identifiers. This is provided by the Enumeration type.

An Enumeration type looks like:

$$\texttt{Enumeration(}id_1\ \texttt{:}\ \ \texttt{Type,}\ id_2\ \texttt{:}\ \ \texttt{Type, ...)}$$

which may be abbreviated to:

$$\texttt{'}id_1, id_2, \ldots\texttt{'}$$

For example,

```
>> import from 'red, green, blue';

>> x := red;
```

```
  () @ Enumeration(red: Type, green: Type, blue: Type)

>> if x = green then stdout << "Strewth";
```

One use of enumeration types is to allow a mechanism for using keywords in signatures. For example,

```
>> import from List MachineInteger, 'first, second';

>> element(l: List MachineInteger, tag: 'first, second'):MachineInteger == {
       tag = first => first l;
       tag = second => first rest l;
       never;
   }

-- called with:
>> element([1,2,3], second);
2 @ MachineInteger

-- also, we can use the apply notation:
>> apply(l:List MachineInteger,tag:'first,second'):MachineInteger==element(l,tag);

-- can be called with
>> apply([1,2,3], second)
2 @ MachineInteger

-- or
>> [1,2,3].second
2 @ MachineInteger
```

In case you were wondering, the syntax for record and union access is implemented this way.

5 Types and other sources of confusion

Most of the features we have seen so far (statements, conditionals, iteration, etc) have direct equivalents in most other programming languages. The major distinguishing feature of Aldor is its type system, which has some analogues in other languages, but no direct equivalent. However, the underlying principles do exist in other languages, and these ideas are quite simple.

The type system in Aldor provides the principal means of extending the language, in this case by adding new libraries. In this section we give a practical introduction to defining types.

As before, you should start a fresh interactive session for this section and include both `aldor` and `aldorinterp`. A first example is in the definition of packages, where we want a group of related operations with a well-defined interface which can then be used by other programs:

```
>> SimpleListStringOperations: with {
    firstOccurence: (List String, String)            -> MachineInteger;
    sort: (List String, (String, String) -> Boolean) -> List String;
} == add {
    firstOccurence(l: List String, s: String): MachineInteger == {
            (l, n) := find(s, l);
            n;
    }

    -- insertion sort.
    sort(l:List String, lt:(String,String) -> Boolean):List String == {
            local insrt(x: String, l: List String): List String == {
                    empty? l or lt(x, first l) => cons(x, l);
                    cons(first l, insrt(x, rest l));
            }
            res:List String := empty;
            for s in l repeat res := insrt(s, res);
            res;
    }
}
```

There are several things to notice here:

- The "`with`" keyword takes a sequence of declarations and turns them into a group of operations (this group is called a *category*).
- The "`add`" keyword takes a sequence of definitions and forms an object that exports definititions, and supplies a way of retrieving the definititions given a declaration. The type of an `add` expression is a category which matches the exports of the `add` body.
- Functions can be nested.

Once we've defined it, we can test it:

```
>> import from SimpleListStringOperations, List String;

>> firstOccurence(["a", "b"], "a")
1 @ MachineInteger
```

It seems natural to ask if there is a way of parameterizing the package by the argument type of the list, so we don't limit ourselves to lists of strings. How do we do this? Well, we can write a function:

```
SimpleListOperations(S: PrimitiveType): with {
        firstOccurence: (List S, S)       -> MachineInteger;
        sort: (List S, (S, S) -> Boolean) -> S;
} == add {
        --- this is the same as before with S replacing
        --- String
}
```

Aside: If we were being truly lazy, we would have called the parameter "`String`*" and left the body unchanged — domain names are just like any other constant, after all.*

The 'S' here is declared as being a "`PrimitiveType`" which is a type (a category in fact) representing all the objects that export an equality operation (this includes Integers, Strings, and many other types in `libaldor`).

It is possible to abstract this further (we do depend on a concrete value "List" that could be replaced by a parameter), but I'll leave that as an exercise.

5.1 Categories

As mentioned above, a "`with`" expression in the definition of `SimpleListOperations` forms something called a *Category*. This is a set of declarations, usually related in some way. In this section, we explore these objects.

As an example, consider a program that compares the efficiency of several different list processing operations. We might want to write this as something like:

```
TestAlgorithms(NthFindSortAndStuff): () == {
        l := [x for x in 1..1000]
        time(firstOccurence, l, 10);
        ...
}
```

where "`NthFindSortAndStuff`" is a (somewhat poorly named) group of operations that includes an implementation of "`firstOccurence`". We then want to pass each of these operations to a function "`time`" which presumably prints a number representing how fast these functions are. In `Aldor`, this is pretty simple:

```
TestAlgorithms(Algorithms: with {
                -- local macro definition
                macro MZ == MachineInteger;
```

```
                firstOccurence: (List MZ, MZ) -> MZ;
                ...
                }
              ): () == {
        import from Algorithms;
        ...
}
```

We can then pass in, for example, `SimpleListOperationsOnMachineIntegers`, `SimpleListOperations(MachineInteger)`, etc. Note that the identifier "`SimpleListOperations`" refers to a function (you can confirm this using the interpreter), and so cannot be passed into this function.

We would like to use this mechanism to identify the common parts of types such as Integer, MachineInteger, MachineFloat, etc... all of which have a complement of arithmetic operations, equality operations, printing operations, etc. In order to do this, we use "`%`" to refer to "this type", so that all these domains satisfy the following category:

```
        with {
                *: (%, %) -> %;
                +: (%, %) -> %;
                -: (%, %) -> %;
                ...
        }
```

In mathematics it is common for several structures to be defined by the same operations, but have different properties (for example, ring, commutative ring, integral domain, etc...). For this reason we treat named categories slightly differently.

```
+++ The category of lattices: structures with a partial ordering
+++ operation such that every pair has a lower and an upper bound
define Lattice: Category == PrimitiveType with {
        -- Partial implies that the function may not always have a value
        < : (%, %) -> Partial Boolean;      ++ partial order
        /\: (%, %) -> %;                    ++ upper bound
        \/: (%, %) -> %;                    ++ lower bound
}
```

Aside: Note the use of descriptions — these are comments introduced by `++` *rather than* `--`*. They are kept with the definition by the compiler, and can be searched using the appropriate tools. A description introduced by* `+++` *refers to the following definition, while a description introduced by* `++` *refers to the preceding definition.*

This defines a category "`Lattice`". As opposed to the anonymous "`with`" expressions above, an object only belongs to this category by assertion, not simply by having the right exports.

In the `SimpleListOperations` examples above, we could restrict `S` to be of type "`TotallyOrderedType`" (a category provided by `libaldor`) so only types with a total order on their values are suitable. This would enable us to export the function:

```
        sort: List S -> S;
```

that would use the total order provided by S. Its implementation would be

```
        sort(l:List S): List S == sort(l, <);
```

where `<` stands for the ordering provided by the type `S`.

5.2 Domains

We now move onto defining new concrete types (one view of categories is as abstract types). This is pretty easy now we know about packages and categories: A domain is simply an `add` body that contains references to "`%`". An example of a simple domain is:

```
>> Wrapped: with {
        wrap:   MachineInteger -> %;
        unwrap: % -> MachineInteger;
                ++ these are such that unwrap(wrap(x)) = x
} == add {
        macro Rep == MachineInteger;
        wrap(n: MachineInteger): % == per n;
        unwrap(x: %): MachineInteger == rep x;
}

>> import from MachineInteger, Wrapped;

-- this is an error
>> wrap(10) + 5
   ^
[L13 C1] #1 (Error) Argument 1 of '+' did not match any possible...
    The rejected type is Wrapped.
    Expected one of:
      -- String
      -- MachineInteger
```

```
-- as is this:
>> unwrap(5)
  ........^
[L14 C9] #1 (Error) Argument 1 of `unwrap' did not match any possible...
    The rejected type is MachineInteger.
    Expected type Wrapped.

-- Compare this with the ``typedef'' construct in C.
```

A few things to note here:

- A domain is exactly the same as a package, except that the signatures of the domain exports include `%`.
- A domain always contains a line "`macro Rep == ...`" which supplies a type that is used to implement the new domain. This is called the representation.
- The macros `rep` and `per` switch between the domain's type and its representation. `rep` yields an object of type `Rep`, `per` yields an object of type `%`.

When writing a domain, there are two important choices to be made:

- The interface to the domain
- How to represent the domain

The interface will typically be a group of categories, plus some explicit operations. In the previous example, we are not a member of any named category, and our explict exports are `wrap` and `unwrap`. The exact categories and explicit operations will tend to be obvious from how you want to use the domain.

The representation of the domain will depend on the interface and how you want to implement the operations. In many cases, the representation will be a record or union type, but there are no restrictions.

In the example above, all the category requests is a domain which can be converted to and from `MachineInteger`. The most efficient way of doing this is to represent values from the domain by MachineIntegers, and use the representation macros to switch between the two.

Of course, we might want to do things differently:

```
Wrapped: with {
        wrap:   MachineInteger -> %;
        unwrap: % -> MachineInteger;
} == add {
        macro Rep == MachineInteger;
        import from Rep;
        wrap(n: MachineInteger): % == per(n-1);
        unwrap(x: %): MachineInteger == rep(x)+1;
}
```

Here a wrapped integer is represented by one less than the original integer. Note that the external interface is unchanged; No user of this domain will be able to detect the change, except perhaps by examining the speed of the wrap and unwrap operations.

In this case we need to import from Rep to get the `+` and `-` operations, and this is generally the case. It is clearer to say "`import from Rep''` rather than from `MachineInteger` as we are thinking of the operations as coming from the representation of the domain, not specifically from `MachineInteger`.

As with packages, there is no problem forming a function so that we can use this for types other than `MachineInteger`.

```
Simple(S: Type): with {
        wrap:   S -> %;
        unwrap: % -> S;
} == add {
        macro Rep == S;
        wrap(n: S): % == per n;
        unwrap(x: %): S == rep x;
}
```

In order to make this type a little more useful, we make it a member of the category `OutputType`. This type means that the elements can be written to output streams such as `stdout`, and many domains in `libaldor` are members of this category, so there is no reason why this domain can't be.

The definition of `OutputType` in `libaldor` is:

```
define OutputType: Category == with {
        <<:     (TextWriter, %) -> TextWriter;
}
```

You can use the interpreter to see what exports a category requires, simply by typing its name:

```
>> OutputType
  () @ Category ==  with <<: (TextWriter, %) -> TextWriter
```

To make our `Simple` domain into an `OutputType`, we simply assert that it is, and add the appropriate definition. Note however that since our new function uses << from S, we need to restrict the parameter `S` to be itself of type `OutputType` rather than simply any `Type`[2]:

```
Simple(S: OutputType): OutputType with {
        wrap:   S -> %;
        unwrap: % -> S;
} == add {
        macro Rep == S;
        wrap(n: S): % == per n;
        unwrap(x: %): S == rep x;

        (out:TextWriter) << (s:%): TextWriter == {
                import from String, Rep;
                -- print a left-paren,
                -- then 's' viewed as an element of S,
                -- then a right-paren.
                out << "(" << rep s << ")";
        }
}
```

The exports of an `add` body are checked against the category, and the compiler will raise an error if any are left out. For example, if any of the functions within the previous `add` body were omitted, the compiler would raise an error.

5.3 Default Operations

Once we have defined a category we find that many operations can be implemented using only the exports of the category, and do dot require infomation about how the domain is represented. For example, in the category `Totally-OrderedType` provided by `libaldor`, once we have a '`<`' operation we can define the `>`, `<=`, `>=`, `min` and `max` operations as[3]:

```
(a:%) > (b:%):Boolean   == ~(a <= b);
(a:%) >= (b:%):Boolean  == ~(a < b);
(a:%) <= (b:%):Boolean  == (a < b) or (a = b);
max(a:%, b:%):%         == { a < b => b; a };
min(a:%, b:%):%         == { a < b => a; b };
```

[2]you may want to compile this domain with `S` declared as `Type` only, to see the reaction of the compiler

[3] ~ means boolean negation in `libaldor`

Can we avoid having to write this code for every `TotallyOrderedType`? The answer is yes, via default operations. These are operations defined inside a with body, which are then inherited by all domains that belong to that category.

Thus, the definition of `TotallyOrderedType` in `libaldor` looks like[4]:

```
define TotallyOrderedType:Category == PrimitiveType with {
        <  : (%, %) -> Boolean;
        >  : (%, %) -> Boolean;
        <= : (%, %) -> Boolean;
        <= : (%, %) -> Boolean;
        min: (%, %) -> %;
        max: (%, %) -> %;
        default {
                (a:%) > (b:%):Boolean   == ~(a <= b);
                (a:%) >= (b:%):Boolean  == ~(a < b);
                (a:%) <= (b:%):Boolean  == (a < b) or (a = b);
                max(a:%, b:%):%         == { a < b => b; a };
                min(a:%, b:%):%         == { a < b => a; b };
        }
}
```

If a domain defines its own `>`, `<=`, `>=`, `min` or `max` operation then the default version will not be used.

5.4 Category Inheritance

You may have noticed the "`PrimitiveType with`" in the preceding category expression. Allowing named categories in "`with`" expressions allows one to say things like "Every `Field` is a `Ring`", and "every `OrderedField` is a `Field`".

For example:

```
define Field: Category == Ring with {
        inv: % -> %;          -- argument is asserted to be nonzero
}

define OrderedField: Category == Join(Field, TotallyOrderedType) with {
        ...
}
```

The "`Join`" operation combines the operations from each of its arguments into a single category. This allows a form of multiple inheritance. It is also possible to list the parents of a category inside the "`with`" body:

[4] the actual definition has a couple more exported functions

```
define OrderedField: Category == with {
        Field;
        TotallyOrderedType;
        ...
}
```

Pick whichever looks tidier.

5.5 Domain inheritance

There are often cases where one domain is an "extended subset" of another domain — for example, the domain of non-negative integers shares many of the properties of rings, although we don't necessarily want all of them (for example, subtraction is no longer a closed operation).

In this case, you can define a domain as above, using "`Integer`" as its representation, but you will quickly find that a lot of the operations will have have rather trivial definitions. For example, "`+`" will be:

```
(a:%) + (b:%):% == per(rep(a) + rep(b));
```

This can become painful quite quickly, so `Aldor` provides a mechanism allowing the definitions from the Integer domain to be included in the exports of the new domain. This is called *domain inheritance*. Using this, the definition of `NonNegativeInteger` will look like:

```
NonNegativeInteger: Join(OutputType, TotallyOrderedType) with {
        -- we use ``+'' to create non-negs ints and coerce to go back
        +:      Integer -> %;
        coerce: % -> Integer;
} == Integer add {
        macro Rep == Integer;
        import from Rep;

        coerce(x: %): Integer == rep x;

        +(x: Integer): % == {
                import from String;       -- for the error function
                x < 0 => error "NNI: negative value";
                per x;
        }
}
```

The important change here is the "`Integer add`", which makes `NonNegative-Integer` inherit `Integer`'s implementations of equality, total ordering, printing, etc. . . .

Note that we still need a `Rep` type, because the implementations of `coerce` and `+` use the `rep` and `per` macros. When inheriting in this fashion, the `Rep` type must match the type mentioned before the `add`.

5.6 Summary of types

In the preceeding section we have defined a lot of inter-related terminology, and a fair number of concepts, so here is a brief recap:

- A category is a collection of declarations of related constants[5] and is formed via either "`with`" or "`Join`".
- A domain defines a group of constants and is formed using "add".
- The special value "`%`" refers to "the type currently being defined".
- Categories may define "default operations". These are operations defined in terms of the other declarations in the category, and are inherited by domains. A domain may over-ride a default implementation.
- Categories inherit operations from their parents, forming a directed graph. If a domain belongs to a particular category, then it aslo belongs to the parents of the category. The parents are given on the left-hand side of the "`with`" construct, or inside the body of the construct.
- Domains may use add-inheritance to satisfy their exports.

5.7 Domains and Category Membership

In many cases, the implementation of an operation can be more efficient if we have additional information about a parameter. For example,

```
gcd(R: EuclideanDomain, x: R, y: R): R == {
        -- insert the euclidean algorithm here
}
```

Now, if R happens to belong to a Field, then everything is a whole lot more trivial:

```
gcd(R: Field, x: R, y: R): R == if zero? x and zero? y then 0 else 1;
```

[5]A constant is a symbol assigned via "`==`", that cannot be assigned a second time once assigned, as opposed to a variable, assigned via "`:=`", which can change its value later.

Can we use the additional information about R in this case?

```
gcd(R: EuclideanDomain, x: R, y: R): R == {
        R has Field => if zero? x and zero? y then 0 else 1;
        -- insert the euclidean algorithm here
}
```

The "`has`" test allows us to see what categories R belongs to at runtime, rather than compile time.

As well as allowing algorithm selection at runtime, `Aldor` also allows you to selectively export operations from a domain or category based on conditions. For example, in our `Simple(S)` example above, we had to restrict `S` to be of type `OutputType` in order to define the printing of objects. Instead, we could be more permissive and conditionaly define `Simple(S)` to be of type `OutputType` only if `S` is also of that type. In order to support this, category expressions may contain *conditional exports*. Similarly definitions, both within "`add` and "`default`" expressions can be conditional:

```
Simple(S: Type): with {
        if S has OutputType then OutputType;
        wrap:   S -> %;
        unwrap: % -> S;
} == add {
        macro Rep == S;
        wrap(n: S): % == per n;
        unwrap(x: %): S == rep x;

        if S has OutputType then {
            (out:TextWriter) << (s:%): TextWriter == {
                import from String, Rep;
                out << "(" << rep s << ")";
            }
        }
}
```

For a more mathematical example: one can build polynomials over any ring to form another ring, but a polynomial built over a field will be a euclidean domain, and consequently have a computable gcd:

```
define UnivariatePolynomial(R: Ring): Ring with {
        if R has Field then EuclideanDomain;
} == add {
        ...
        if R has Field then {
                gcd(x: %, y: %): % == {
                      -- insert your favorite Euclidean algorithm here.
```

```
                }
        }
        ...
}
```

NB: Conditionals can be very useful, but overuse can lead to very confusing code.

6 Interfacing with other languages

One of the design goals of `Aldor` was that it should be possible to both call and be called from libraries written in other languages. In order to do this, there is a builtin "Type" called `Foreign`, which can be viewed as an interface to other languages.

6.1 Using functions from other libraries

For example, most C libraries contain a function called `unlink`, which deletes a file and returns a code indicating successful or unsuccessful completion. If we wanted to use this function from `Aldor`, then we do this:

```
>> import { unlink:  String -> MachineInteger } from Foreign C;
```

This tells the compiler that there is a function named unlink that is available. The compiler will then treat this name specially, and use the C calling conventions in order to call it. Consequently, calling it is really easy:

```
>> confirmAndRemove(name:String):MachineInteger == {
        import from TextWriter, TextReader, Character;
        stdout << "Are you sure you want to remove " << name << "? ";
        answer:String := << stdin;
        answer := map(lower)(answer);           -- make lower case
        answer = "yes" or answer = "y" => unlink name;
        -1;
}

>> confirmAndRemove "foo"
Are you sure you want to remove foo? no
-1 @ MachineInteger
```

Aside: Typing "yes" will not work in the interpreter as it cannot dynamically link to new C functions.

```
UNLINK(2)           Linux Programmer's Manual           UNLINK(2)

NAME
       unlink - delete a name and possibly the file it refers to

SYNOPSIS
       #include <unistd.h>

       int unlink(const char *pathname);

DESCRIPTION
       unlink  deletes  a  name from the filesystem. If that name
       was the last link to a file and no processes have the file
       open  the  file  is  deleted and the space it was using is
       made available for reuse.

       If the name was the last link to a file but any  processes
       still have the file open the file will remain in existence
       until the last file descriptor referring to it is  closed.

       If  the  name  referred  to  a  symbolic  link the link is
       removed.

       If the name referred to a socket, fifo or device the  name
       for it is removed but processes which have the object open
       may continue to use it.

RETURN VALUE
       On success, zero is returned.  On error, -1  is  returned,
       and errno is set appropriately.
```

Figure 1: A Manual page for some C routine

Note that constants work just like functions and can be imported from `Foreign` too.

Aside: Since the underscore is used as an escape character in `Aldor`*, rather than backslash as in C, you must replace every underscore appearing in a C name by '`__`' in its* `Aldor` *counterpart.*

As well as `String`, other `libaldor` types may be used in signatures (*i.e.* prototypes) of foreign functions:

libaldor	C
Boolean	long
MachineInteger	long
SingleFloat	float
Character	char
String	char*
PrimitiveMemoryBlock	char*
PackedPrimitiveArray Byte	char*
PackedPrimitiveArray Character	char*
PackedPrimitiveArray SingleFloat	float*
PackedPrimitiveArray DoubleFloat	double*
Pointer	void*
PrimitiveArray	void**
PrimitiveArray MachineInteger	long*
File	FILE*
GMPFloat	mpf_t
GMPInteger	mpz_t

Strings in `libaldor` are represented by zero-terminated strings, as in C.

6.2 Exporting functions to C

As you might guess from the previous section, exporting functions to C is pretty simple. In fact, the syntax is very familiar:

```
export {
        is__prime: MachineInteger -> Boolean;
} to Foreign C;

is__prime(x: MachineInteger): Boolean == ...
```

This can be compiled into an object file using "`aldor -fo`", and then linked in the usual way. You will need to link the final program to the `Aldor` runtime library — the easiest way to find out what this is is to compile a stand-alone program with -v and see what libraries it uses. Note that exporting constants to C requires more work, as you have to ensure that the exporting file is initialized properly.

7 Exceptions

7.1 Introduction

What happens when something goes wrong in a function? For example, your system detects an error in some parameters, or that it can't perform a particular operation (eg. opening a file).

Using normal call and return code, you have two options:

The function can cause the program to halt, eg. by a call to "`error`", or it can return a `Partial` type which can indicate a failed call.

The former may be a little extreme, especially if the program is in some way interactive.

The latter option then causes every function that calls it to have to be aware of the failure case. This makes the funcions's interface more complicated, and may sometimes be impossible. For example, consider the division exported by `ArithmeticType` — it has the signature `(%, %) -> %` and is expected to yield a value. Consequently we aren't allowed to return a union object even if we wanted to (we don't, because unions can be quite costly to create, when compared with the time needed to do a division).

A more consise way of signalling error conditions is provided by the exceptions mechanism (for those familiar with C++ exceptions, this should be unsurprising). There are four parts to the mechanism:

1. Throwing (or raising) an exception
2. Catching exceptions that we are interested in
3. Declaring which exceptions may be raised by an expression
4. Defining exceptions

The first two are fairly standard, and the syntax is similar to that of many other languages (especially C++). The last two have their roots in the `Aldor` type system.

7.2 Throwing Exceptions

The syntax for raising an error is:

```
throw <E>
```

where `<E>` is an expression evaluating to an exception object. When an exception is thrown, execution of the current function halts, and control is passed to the most recent handler (note that this uses dynamic scope for determining “most recent”). The type of an except expression is “Exit” (but see later) — ie. no value is created by the expression, but control does not pass the expression (compare this with the return statement, for example). An except statement is therefore a way of causing a function to terminate abnormally.

7.3 Catching Exceptions

To catch an exception the syntax is:

`try` *Expr* `catch` *id* `in` *handler* `always` *stmts*

where *Expr* is the expression we are protecting, *id* is an identifier, *handler* is an (optional) error handler and *stmts* are some (optional) statements to do any resource freeing on abnormal exits.

When a handler is executed, the *id* is bound to whatever exception was raised. The handler block is then evaluated.

Typically it will look something like:

```
try ... catch E in {
        E has ZeroDivideType => ...
        E has BadReductionType => ...
        E has FileExceptionType => ...
        true => throw E;
        never;
}
```

Where each item on the right hand side of the `has` denotes an exception type. The last two lines are currently required, but may go away at some point.

Each branch of the handler block (the parts after “`=>`”) should evaluate to the same type as that of the expression protected by the handler.

This is so that one can do:

```
n := try divide(a, x) catch E in { E has ZeroDivideType => 22; ... }
```

which will attempt the division, and if successful assign the result to "`n`", otherwise if a division by zero exception is raised, then "`n`" will have the value 22.

After the handler has been executed, any forms in the "`always`" part of the try expression are evaluated. These are guarenteed to be evaluated even if the handler itself raises an exception.

The typical reason to use an always block is to deallocate any resources the function may have allocated, for example:

```
f := open(file);
try doWonderfulThings(f, data) always close(f);
```

Note that the '`catch` *id* `in ...`' part is also optional providing an always part is supplied.

This will ensure that the file is always closed regardless of what exceptions are thrown by the call to `doWonderfulThings`.

7.4 Exception types

It is possible to declare what exceptions are thrown by a particular expression. This is done using the "`throw`" keyword as an infix operator (the two uses are rarely confused). The operator takes two arguments, a base type and a (possible empty) comma-separated list of exception objects. Typically, the throw keyword is applied to the return types of functions to indicate which exceptions they can raise, for example:

```
foo: (...) -> ... throw (x1,x2,x3...)
```

This indicates that foo may only raise exceptions x1,x2,x3, etc. The programs behaviour is undefined if other exceptions are raised. To indicate that a function raises no exceptions, the tuple should be empty. If there is no 'throw' clause on the return type, then the compiler assumes that any exception may be raised by the function. This does lead to some unsafe code — for example:

```
justDie(): MachineInteger == throw ZeroDivide;

badIdea(): MachineInteger throw () == justDie();
```

Here 'badIdea' indicates that it will not raise an exception, while 'justDie' will always raise ZeroDivide. The compiler may warn the user in this situation, but not at the moment.

The compiler will however check if any exceptions are explicitly raised that do not satisy the functions signature, for example:

```
foo(): MachineInteger throw ZeroDivide == {
        throw FileError;
}
```

will not compile. This also works within try blocks:

```
bar(): () throw Ex1 == {
        try zzz() catch E in {
                E has ExA => throw ZeroDivide; --- error
                E has ExB => throw Ex1; --- OK
                true => throw E --- error
                never
        }
}
```

Here, the final case has to be modified to return the exception Ex1, or deleted so the program will halt on the "never" statement.

The 'throw' qualifier on types works just like any other type constructor, and so you can use it as part of a type as normal:

```
        foo(fn: MachineInteger -> MachineInteger throw ()): () == ...
```

indicates that the argument to foo must be a function that never raises an exception. Naturally, there are only a few places where it makes sense to use these types.

NB: I simplified things earlier when I said that the type of an `throw` statement was `Exit` — the actual type of "`throw X`" is "`Exit throw X`". This indicates that flow of contol stops, but the exception `X` may be raised. Exactly what we wanted, surprisingly enough.

7.5 Defining Exceptions

An exception definition is made up of two parts — a category definition and a domain definition. The category definition provides a means to specify related exceptions (so that 'ZeroDivideType' may inherit from 'ArithmeticErrorType' for example), and the domain definition provides a mechanism for creating the exception.

For example,

```
define ZeroDivideType: Category == ArithmeticErrorType with;
ZeroDivide: ZeroDivideType == add;
```

If ZeroDivide is defined this way, then any handler with "E has ArithmeticErrorType" will also catch ZeroDivide exceptions.

This mechanism also allows one to create parameterized exceptions:

```
define ZeroDivideType(R: Ring): Category == ArithmeticErrorType with;
ZeroDivide(R: Ring): ZeroDivideType R == add;
```

and to have values defined within the exception:

```
define FileExceptionType: Category == ArithmeticErrorType with {
        name: String;
}

FileException(vvv: String): FileExceptionType == add {
        name: String == vvv;
}
```

In fact, there is a myriad of ways to insert values into exceptions:

```
foo():() == {
    ...
    throw (add {name: String == <calculation>})@FileExceptionType;
    ...
}
```

is one, which has the single advantage that "name" will only be calculated if it is actually used. Defining a 'lazy' version of FileException is a much better thing to do in this situation.

Another example that keeps the add definition simple, but sort of loses it with respect to the number of objects defined.

```
define FileExceptionType: Category == ExceptionType with {
        name(): String;
}

define FileExceptionType(vvv: String): Category == FileExceptionType with {
        default name(): String == vvv;
}

FileException(v: String): FileExceptionType v == add;
```

8 Introductory Assignments

This section provides a few example exercises which provide a feel for how to write programs in `Aldor`. They are arranged in approximate order of difficulty.

8.1 Hello world

Write a function to print the words "hello world" on the console.

Feel good about it.

8.2 Factorial

Write a function that calculates the factorial function, defined by:

$$n! = \begin{cases} 1 & \text{if } n < 2 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

The function should work on arbitrary precision integers.

This can be done in two ways: by recursion, or by iteration. The definition above is ideal for recursion, while the rule $n! = 1 * \ldots * (n-1) * n$ is appropriate for the iterative version.

8.3 Summing a list

Write a function that returns the sum of all the elements in a list. The program should be as general as possible, especially if you have read the section on types.

8.4 Summing an arbitrary sequence

The previous exercise produced a value from a list. What about arrays, trees, and other aggregates? Write a function that sums all the elements from a generator, and returns that sum. Review the section on collections if necessary.

8.5 Reducing things

The reduce function (in Axiom it is called "/") takes an associative function and a sequence and repeatedly applies the function to the sequence. It is defined as: reduce(f, l) = first l, if the sequence contains 1 element reduce(f, l) = f(first l, reduce(f, rest l))

Write both recursive and iterative versions.

As in the "sum" example, modify either program to work with an arbitrary sequence.

Do exercise 8.3 again.

8.6 Tails

Write a function `tails(S:Type, l:List S): Generator List S == ...` that returns a generator for the sequence `l, rest l, rest rest l,...` and that ends when all the nonempty tails have been yielded. For example:

```
l:List MachineInteger := [1,2,3,4];
for t in tails(MachineInteger, l) repeat stdout << t << newline;
```

should print

```
[1,2,3,4]
[2,3,4]
[3,4]
[4]
```

8.7 Generators

Re-write the "`both`" example so that the lists do not have to be the same size. When one list is exhausted, simply use the remaining elelements from the other list. Exercise 8.6 may be useful here.

9 Type Assignments

There are lots of possible domains that you should be able to implement. Depending on inclination, you could try series, complex numbers, quaternions, simple polynomials, trees, heaps, tables, etc. If none of the exercises below looks interesting, feel free to roll your own.

9.1 Sorted Lists

Use add-inheritance to build a domain of sorted lists. This is a list whose elements are guarenteed to be in sorted order. Many operations are identical to those from `List`, but a few will differ, for example, construction. The domain should be of category `BoundedFiniteLinearStructureType`, and any others you think are useful. Since there is already a type `SortedList` in `libaldor`, use a different name.

9.2 Sorting Package

Write a package exporting a few sort algorithms for arrays of integers. For example insertion sort, bubble sort and so on.

Modify this to work with arrays. Generalise as far as you can.

9.3 Substrings

In this exercise you will complete the definition of a `SubString` domain, whose objects are represented by a string, a starting position within the string, and a substring length. The `SubString` domain can then be used to write non-copying string processing functions.

This example also introduces the equivalence between the expression "`x::T`" in `Aldor`, and the application "`coerce(x)@T`." This kind of mechanism allows programmers to overload the behavior of certain keywords in `Aldor`.

```
--- Substrings.

#include "aldor"

-- (a) Fill in the definitions for substring, coerce, <<.

-- substring creates a new substring object from a string,
-- a starting position, and a length.

-- coerce copies the characters from the string
-- into a new string.

-- << is the function for printing the substring
-- which is required by BasicType.

-- sample provides a default substring.

SubString: OutputType with {
        substring: (String, MachineInteger, MachineInteger) -> %;
                ++ construct a substring of a string

        coerce:    % -> String;
                ++ copies the substring into a new string
}
== add {
        macro MZ == MachineInteger;
        macro Rep == Record(str: String, start: MZ, len: MZ);
        import from Rep;

        substring(str: String, start: MZ, end: MZ): % == ...

        coerce(substr: %): String == ...

        (out: TextWriter) << (sub: %): TextWriter == ...
}

-- (b) Write a function which takes a string and splits it
--     into a list of words from the string which are delimited
--     by spaces and newlines.

-- (c) Write a function which takes a string and generates
--     the words from the string.
```

You may want to try changing the representation of this type to a `String`, and comparing the relative execution times of functions (b) and (c) using each version.