Aldor User Guide

by Aldor.org

Aldor User Guide

Substantial portions of this manual have been published earlier in the volume "Axiom Library Compiler User Guide," The Numerical Algorithms Group 1994, `ISBN 1-85206-106-5`.

Aldor was originally developed, under the working name of $A^\sharp$, by the Research Division of International Business Machines Corporation, Yorktown Heights, New York, USA.

Acknowledgements

Aldor was originally developed at IBM Yorktown Heights, in various versions from 1985 to 1994, as an extension language for the AXIOM computer algebra system. Aldor was further refined and extended at the Numerical Algorithms Group at Oxford, before the formation of Aldor.org in 2002.

The principal designer of the language was Stephen Watt, now Professor of Computer Science at the University of Western Ontario. Many people have worked on the compiler, its documentation and the associated libraries, including Gerald Baumgartner, Dave Bayer, Peter Broadbery, Manuel Bronstein, Ronnie Brown, Florian Bundshuh, Bill Burge, Yannis Chicha, Robert Corless, George Corliss, Tim Daly, James Davenport, Mike Dewar, Samuel Dooley, Martin Dunstan, Robert Edwards, Marc Gaetano, Patrizia Gianni, Teresa Gomez-Diaz, Stephen Gortler, Johannes Grabmeier, Vilya Harvey, Ralf Hemmecke, Pietro Iglio, Tony Kennedy, Larry Lambe, Ian Meikle, Michael Monagan, Marc Moreno-Maza, Scott Morrison, Bill Naylor, Mike Richardson, Simon Robinson, Philip Santas, Jonathan Steinback, Robert Sutor, Themos Tsikas, Barry Trager, Mike West, and Knut Wolf.

Thanks also to the management at IBM and NAG, in particular Brian Ford, Steve Hague, Shmuel Winorgrad, Bill Pulleyblank, Marshall Schor and Richard Jenks for their cooperation in bringing Aldor out of the laboratory and into the real world.

Preface

Aldor is a programming language that attempts to achieve power through the uniform treatment of all values. Rather than build a language by adding features, we have tried instead to build a language by removing restrictions. While the design of Aldor emphasizes generality and composibility, it also emphasizes efficiency. Usually these objectives seem to pull in different directions. An achievement of Aldor's implementation is its ability to attain both simultaneously.

Aldor is not at its foundation an object-oriented language. Instead, object semantics are reconstructed from the primitive treatment of functions and types as first-class values. Similarly, aspect-oriented programming arises as a natural use of general language primitives. While the initial rôle of Aldor was to replace the compiler component of the computer algebra system AXIOM, Aldor is not a reimplentation of the AXIOM programming language. Rather, Aldor reconstructs the essential aspects of the AXIOM programming language from more primitive notions.

Aldor has been, over the period 1995-2001, available from the Numerical Algorithms Group (NAG) as part of the commercial AXIOM system. Over this period, Aldor's users began using it more and more outside of this original context, to the point where now most Aldor code is unrelated to AXIOM.

It is now appropriate that Aldor have its own means of distribution for those who wish to use it in a general context, and `Aldor.org` has been formed for this purpose. The Numerical Algorithms Group has graciously consented to allow free distribution of Aldor this way.

Bringing Aldor from a gleam in the mind's eye to a concrete compiler has been a substantial task. Considerable program analysis and optimization is required to reduce high-level source programs to efficient machine code. For the current release, the source of the compiler is approximately 135,000 lines of code, not including the run-time system, base library or other associated software.

Here we have described Aldor 1.0, the first release of Aldor independent of AXIOM. Those who have used Aldor earlier will note that the present document has been adapted to refer to a new library, `libaldor`. This library has been used as there is now a considerable body of Aldor code based upon it. Manuel Bronstein and Marc Moreno Maza are to be thanked for having invested considerable efforts in its development. Marc Moreno Maza and Yannis Chicha have updated all the examples in this document to work with `libaldor`.

Numerous individuals have contributed to Aldor over its development at IBM Research, the Numerical Algorithms Group, and elsewhere. These contributors should be listed in the acknowledgements section of this document. Particular thanks are due to Martin Dunstan who served as Aldor's steward at the Numerical Algorithms Group. It has been a true pleasure to work with such a collegial and insightful group of people.

London, Ontario *SMW*
January, 2002

Summary Contents

Contents

Figures

PART I

A brief overview of Aldor

CHAPTER 1

Introduction

1.1 What is Aldor?

The original motivation for Aldor came from the field of computer algebra: to provide an improved extension language for the AXIOM system.

The desire to model the extremely rich relationships among mathematical structures has driven the design of Aldor in a somewhat different direction than that of other contemporary programming languages. Aldor places more emphasis on uniform handling of functions and types, and less emphasis on a particular object model. Aldor is an acronym, standing for **A** **L**anguage for **D**escribing **O**bjects and **R**elationships.

The primary considerations in the formulation of Aldor have been generality, composibility and efficiency. The Aldor language has been specifically designed to admit a number of important optimizations, allowing compilation to machine code whose efficiency is frequently comparable to that produced by a good C or Fortran compiler.

Aldor is unusual among compiled programming languages, in that types and functions are *first class*: that is, both types and functions may be constructed dynamically and manipulated in the same way as any other values. This provides a natural foundation for both object-oriented and functional programming styles, and leads to programs in which independently developed components may be combined in quite powerful ways.

Two novel features of Aldor are *dependent types*, which allow static checking of dynamic objects, and *post facto type extensions*, which allow complex libraries to be separated into decoupled components.

The Aldor compiler described in this Guide can produce:

- stand-alone executable programs,
- object libraries in native operating system formats,

- portable byte code libraries,
- C source.

The object libraries produced by the Aldor compiler can be linked with one another, or with C or Fortran code, to form application programs. The byte code libraries can be interpreted, and are used by the compiler for inter-file optimization.

The Aldor distribution includes:

- an optimising compiler for the Aldor language,
- an interpreted, interactive environment for the *same* language,
- libraries providing data structures and mathematical abstractions,
- library bindings for standard tools including the NAG Fortran Library,
- sample programs for symbolics, numerics and graphics.

The Aldor compiler has been designed for portability and runs in many different environments. Code generated by Aldor will run on 16, 32 and 64-bit architectures. For an up-to-date list of available implementations, please visit the official Aldor website: `http://www.aldor.org`.

1.2 Compiling and running a single file

The first thing many people want to do is compile and run a simple test file. This section shows how to do this.

We start with an Aldor source file, "`sieve.as`", containing the simple program shown in Figure 1.2.

To compile this file and run the resulting executable program, use the following commands:

```
% aldor -Fx -laldor sieve.as
% ./sieve

There are 4 primes <= 10
There are 25 primes <= 100
There are 168 primes <= 1000
There are 1229 primes <= 10000
There are 9592 primes <= 100000
There are 78498 primes <= 1000000
```

In this example "`%` " is the operating system command line prompt and should not be typed. On most platforms the command to run the Aldor compiler is "`aldor`".

The "`aldor`" command takes the source file "`sieve.as`" and produces a file of machine code which can perform the computation. The executable program is named according to the operating system's usual conventions: for instance, "`sieve`" on Unix, or "`sieve.exe`" on Windows. Once compiled, the new program can be used in the same way as other executable programs for the given operating system.

```
--
-- sieve.as: A prime number sieve to count primes <= n.
--
# include "aldor"
# include "aldorio"

import from Boolean, MachineInteger;

sieve(n: MachineInteger): MachineInteger == {
        isprime: PrimitiveArray Boolean := new(n, true);

        np := 0;
        for p in 2..n | isprime p repeat {
                np := np + 1;
                for i in 2*p..n by p repeat isprime i := false;
        }
        np
}

for i in 1..6 repeat {
        n  := 10^i;
        stdout << "There are " << sieve n << " primes <= " << n;
        stdout << newline;
}
```

Figure 1.1: An Aldor program.

Command line options control the behaviour of the compiler. For example, the option "`-Fx`" in the previous example directs the compiler to produce an executable file. Also, the option "`-laldor`" directs the linker to compile the file using "`libaldor.a`".

There are many available command line options, regulating different aspects of the compiler's actions. They allow you to control the details of what the compiler actually does. Here we point out a few of the most important options — the rest are described in detail in chapter 23.

Keep in mind that you do not need to remember very much. The only option you really need to know is "`-help`", which gives help. The command is:

```
% aldor -help
```

Another thing to keep in mind is that you can make your programs run *much* faster by asking the compiler to optimize them. The "`-O`" option tells the compiler to do this:

```
% aldor -O -Fx -laldor sieve.as
```

Depending on the way in which Aldor has been installed on your computer, you may need to set some system-specific variable or macro so that the compiler can find its libraries. The value for this will depend on where Aldor is installed. For example, on one of our local Unix systems, this is achieved by setting the "environment variable" "`ALDORROOT`" to

```
/usr/local/aldor
```

To be able to use Aldor on this particular system, one might put the following commands in a (Bourne or Korn shell) initialization file:

```
ALDORROOT=/usr/local/aldor
PATH=$ALDORROOT/bin:$PATH
export ALDORROOT
export PATH
```

Please refer to your system administrator for details of the corresponding setup on your particular computer system.

1.3 This guide

This guide describes the Aldor programming language, a compiler and an interpreter, and other related software.

Part I: The first two chapters provide a quick, informal introduction to Aldor.

Chapter 1 provides an introduction and indicates how to compile and run simple programs. It gives a very brief description of what Aldor is and what the compiler can do. Section 1.4 on page 7 explains how to report problems.

Chapter 2 discusses a number of (mainly) very simple programs.

Part II: The next chapters provide a guide to the Aldor programming language.

Chapters 3 to 15 present in detail the various aspects of the language and provide a number of illustrative examples.

Other chapters in this Guide can also be useful in learning about the language. Additional programming examples are discussed in chapter 21. The formal language syntax is given in chapter 22.

Part III: The next five chapters serve as a guide to the Aldor compiler and related software.

Chapter 16 explains how to interpret and control messages from the compiler.

Chapter 17 describes how to build an Aldor program from several separately compiled files.

Chapter 18 shows how to use the Aldor compiler interactively, to compile and evaluate a line of code at a time.

Chapter 19 shows how to write Aldor programs which call C programs and *vice versa*.

Chapter 20 shows how to write Aldor programs which call Fortran programs and *vice versa*.

Part IV: The next chapter provides some examples to help learn the language.

Chapter 21 provides a number of detailed sample programs. This includes examples which range from trivial half-page programs to complete applications. These provide concrete illustrations of how to use the various aspects of the programming language.

Part V: The remaining chapters provide reference material, and are not intended to be read sequentially.

Chapter 22 is a formal description of the language syntax.

Chapter 23 provides a detailed description of the "`aldor`" command. It describes the types of files, all the options, and the environment variables understood by the compiler.

Chapter 24 discusses the use of the back-end compiler and linker driver `unicl`.

Chapter 25 lists all the messages which the compiler can produce. The names of the messages are also listed so you can turn off specific messages if you wish.

1.4 Reporting problems

If you discover an error in the Aldor compiler, libraries, or companion software we want to know about it so we can fix it.

When reporting a problem, please supply the precise compiler version and have a file that demonstrates the problem. To determine your compiler version, use the "`-v`" option to cause the Aldor compiler to operate verbosely. The first output line will contain the compiler version. For example,

```
% aldor -v file.as
Aldor version 1.0.0 for LINUX(glibc2.2)
               ld in sc sy li pa ma ab ck sb ti gf of pb pl pc po mi
 Time    0.1 s  0  6  0  0  0  0  0  0  0  0 94  0  0  0  0  0  0  0 %
 Alloc  2523 K  0 .3 .3  3 .2 .2 .2 .1  0 .2 93  1 .2  1  0 .0 .0 .0 %
 Free    924 K  0 .1 .1 .0  1 .0 .3 .1  0 .1 92  1  1  4  0 .0  0  2 %
 GC      313 K  0  0  0  0  0  0  0  0  0  0100  0  0  0  0  0  0  0 %

 Source   90 lines,  33750 lines per minute
 Lib    6533 bytes,  2323syme 1241foam 36fsyme 1080name 101kind 1034file 266lazy 246type 2inl 32twins 2ext 2doc 9id 3macros
 Store  2280 K pool, 2523K alloc - 924K free - 313K gc = 1286K final
```

There are two ways to report a problem:

- (recommended) Use the `aldorbug` tool supplied with any Aldor distribution to send a description of the problem and all the necessary files to reproduce it.

or

- Send an email with a description of the problem and all the necessary files to reproduce it to `bug-report@aldor.org`.

CHAPTER 2

Some simple programs

Perhaps the easiest way to get a feeling for a programming language is to read a few programs. This chapter presents simple programs and uses them as a departure point to discuss some of the basic ideas of Aldor.

Two main options are available to readers after completing this chapter. Those who prefer a structured approach may choose to progress through the development in part II. Those preferring to learn by example may want to skip ahead to chapter 21, where they will find extended examples of more advanced programming techniques in the form of further annotated programs, and refer to part II only as necessary.

2.1 Doubling integers

```
#include "aldor"

double(n: AldorInteger): AldorInteger == n + n
```

Figure 2.1: Simple program 1

The first program is one which doubles integers. This program illustrates a number of things:

1. The Aldor language is itself almost empty. This allows libraries to define their own environments all the way down to such basic questions such as what an integer ought to be. Therefore, almost all programs begin with an "`#include`" line to establish a basic context. The "`#include`" line in this example provides a context in which "`Integer`" has a specific meaning, provided by the stand-alone `libaldor` library. Actually the identifier "`Integer`" is a convenient "`macro`" (see chapter 13) for "`AldorInteger`". Please refer to the `libaldor` library documentation for more details about

"Integer" and "AldorInteger".

2. The symbol "==" indicates a definition — in this case a definition of a function named "double".
3. The function has two declarations using the syntax ": Integer". Names indicating values (variables, parameters, *etc.*) may each contain values of only a specific *type*. The first declaration in this program states that the parameter n must be an Integer. The second asserts that the result of the function will also be an Integer. (The type of the function itself is represented as "Integer -> Integer"; a name and type together are called a *signature*, as in "double: Integer -> Integer".)
4. The declarations cause the exports from the type Integer to be visible. Typically, a type exports special values (such as 0 and 1) and functions on its members. In this example, the name "+" has a meaning as an exported function from Integer.
5. The body of the function double is the expression "n + n". The value of this expression is returned as the result of the function. It is not necessary to use an explicit "return" statement, although it is permitted. This turns out to be very convenient when many functions have very short definitions, as is normal with abstract data types or object-oriented programs.

2.2 Square roots

```
#include "aldor"

-- Compute a square root by six steps of Newton's method.
-- This gives 17 correct digits for numbers between 1 and 10.

DF ==> DoubleFloat;

miniSqrt(x: DF): DF == {
        r := x;
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r
}
```

Figure 2.2: Simple program 2

Our second program illustrates several more aspects of the language:

1. Comments begin with two hyphens "--" and continue to the end of the line.
2. Abbreviations ("macros") may be defined using "==>". The line

    DF ==> DoubleFloat;

causes "DF" to be replaced by "DoubleFloat" wherever it is used.

3. A function's body may be a compound expression. In this example the body of the function is a sequence consisting of eight expressions separated by semicolons and grouped together by braces. These expressions are evaluated in the order given. The value of the last expression is the value of the sequence, and hence is the value of the function.
4. The semicolons separate expressions. It is permitted, but not necessary, to have one after the last expression in a sequence.
5. Variables may be assigned values using ":=".
6. The variable "r" is local to the function miniSqrt: it will not be seen from outside it. Variables may be made local to a function by a "local" declaration or, as in this case, implicitly, by assignment.
7. In this function the variable "r" contains double precision floating point values. Since this may be inferred from the program, it is not necessary to provide a type declaration.

2.3 A loop and output

```
#include "aldor"
#include "aldorio"

factorial(n: Integer): Integer == {
        p := 1;
        for i in 1..n repeat p := p * i;
        p
}

import from Integer;

stdout << "factorial 10 = " << factorial 10 << newline;
```

Figure 2.3: Simple program 3

The third program has a loop and produces some output. Things to notice about this program are:

1. This example has expressions which occur at the top-level, outside any function definition. This illustrates how the entire source program is treated as an expression sequence, which may (or may not) contain definitions among other things. This entire source program is treated in the same way as a compound expression forming the body of a function: it is evaluated from top to bottom, performing definitions, assignments and function calls along the way.
2. As we saw in a previous example, the declarations ": Integer" suffice to make the exports from Integer visible within the factorial function. This gives meaning to "1", "*" and "..".
   These declarations do not, however, cause the exports from Integer to be visible at the top-level of the file, outside the function factorial.

This conservative behaviour turns out to be quite desirable when writing large programs, since adding a new function to a working program will not pollute the name space of the program into which it is inserted.

In order to be able to use the exports of `Integer` at the top-level of the file, we have used an "`import`" statement. Importing `TextWriter` allows the use of `stdout`, `String` is needed for `"factorial 10 = "` and `Character` allows the use of `newline`.

3. The last line of the example produces some output. The general idea is to use the infixed name "`<<`" to output the right-hand expression via the left-hand text writer.

   Syntactically, "`<<`" groups from left to right so a minimum number of parentheses are needed: the line in the example is equivalent to
   ```
   ((stdout << "factorial 10 = ") << factorial 10) << newline;
   ```
4. The last line is simple but refers to many things. We shall say exactly where each of them comes from:
   - The name "`stdout`" is a `TextWriter`.
   - The expression "`"factorial 10 ="`" is a `String` constant.
   - "`newline`" is a `Character`.

   All the above are visible by virtue of the "`#include "aldorio"`" statement at the beginning of the example.
   - We have already seen where "`factorial`" and "`10`" come from.
   - This leaves "`<<`". There are three uses, and each use refers to a different function. The first one takes a string as its right (second) argument and comes from the `#include "aldorio"` statement. The second one takes an `Integer` and is imported along with the other operations in with the "`: Integer`" declaration. The last one takes a `Character` and also comes from the `#include "aldorio"` statement.
5. Let us look more closely at the use of the `factorial` function in the last line: "`factorial 10`". No parentheses are needed here because the function has only a single argument. If the function took two arguments, *e.g.* "`5, 5`", or if the argument were a more complicated expression, *e.g.* "`5 + 5`", then parentheses would be needed to force the desired grouping.
6. A word of caution is necessary here: the manner of output is defined by the particular library, not the language. The form of output in this example is appropriate when using "`#include "aldorio"`" but may not work in other contexts.

The difference between `#include "aldor"` and `#include "aldorio"` might not be clear yet, here is an explanation. `#include "aldor"` is the general include statement that will allow the types and functions from `libaldor` to be visible. It is necessary to use it (or an equivalent library) in order to write programs. `#include "aldorio"` is merely a convenience which will automatically import types used for input/output (and, in particular, printing values to the standard output).

2.4 Forming a type

```
#include "aldor"

MiniList(S: OutputType):  MiniListType(S) == ...
```

Figure 2.4: Simple program 4 — Skeleton

The fourth example shows `MiniList`, a type-constructing function.

We will defer showing the body of the function for a moment, until we have had a first look at the definition.

The `MiniList` function will provide a simple list data type constructor, allowing lists to be formed with brackets, for example `[a, b, c]`. All the elements of the list must belong to the same type, `S`, the parameter to the function.

This is a simple example of how one might use `MiniList`:

```
MI ==> MachineInteger;
square(n: MI): MI == {
        sql: MiniList MI := [1, 4, 9, 16, 25];
        if n < 1 or n > 5 then error "Value out of range";
        sql.n  -- the (n)th component of sql
}
```

The definition of `MiniList` is just like the definitions we have seen in the previous examples:

- `MiniList` accepts a parameter, `S`, which is declared to belong to a particular type — in this case "`OutputType`".
- `MiniList` returns a result which is declared to belong to a particular type — in this case to the type "`MiniListType(S)`".
- The body of the function `MiniList` is an expression to compute the return value, given a value for the parameter.

The name `OutputType` is meaningful by virtue of the "`#include`" line.

`OutputType` is a type whose members are themselves types. The same is true for `MiniListType(S)`. A type such as this, whose members are types, will be called a *type category*, or simply a *category* where no confusion can arise.

What we have seen implies that `MiniList` is a function which takes a type parameter and computes a new type as its value.

Now that we have had the bird's eye view, it is time to take a second look at the function. The complete definition is given in Figure 2.4.

```
#include "aldor"

define MiniListType(S: OutputType): Category == with {
		empty: %;
		empty?: % -> Boolean;
		bracket: Tuple S -> %;
		bracket: Generator S -> %;
		generator: % -> Generator S;
		apply: (%, MachineInteger) -> S;
		<<: (TextWriter, %) -> TextWriter;
}

MiniList(S: OutputType):  MiniListType(S) == add {
	Rep == Union(nil: Pointer, rec: Record(first: S, rest: %));
import from MachineInteger, Boolean, Rep;

	local cons (s:S,l:%):% == per(union [s, l]);
	local first(l: %): S   == rep(l).rec.first;
	local rest (l: %): %   == rep(l).rec.rest;

	empty: %               == per(union nil);
	empty?(l: %):Boolean   == rep(l) case nil;

	[t: Tuple S]: % == {
		l: % := empty;
		for i in length t..1 by -1 repeat
			l := cons(element(t, i), l);
		l
	}
	[g: Generator S]: % == {
		r: %  := empty; for s in g repeat r := cons(s, r);
		l: %  := empty; for s in r repeat l := cons(s, l);
		l
	}
	generator(l: %): Generator S == generate {
		while not empty? l repeat {
			yield first l; l := rest l
		}
	}
	apply(l: %, i: MachineInteger): S == {
		while not empty? l and i > 1 repeat
			(l, i) := (rest l, i-1);
		empty? l or i ~= 1 => error "No such element";
		first l
	}
	(out: TextWriter) << (l: %): TextWriter == {
		empty? l => out << "[]";
		out << "[" << first l;
		for s in rest l repeat out << ", " << s;
		out << "]";
	 }
}
```

Figure 2.5: Simple program 4 — Details

A few points will help in understanding this program:

1. The first thing to note is that the new type constructor is defined as a function `MiniList` whose body is an "`add`" expression, itself containing several function definitions. It is these internal functions of an "`add`" function which provide the means to manipulate values belonging to the resulting types (such as `MiniList(Integer)` in this case).
2. This program uses various names we have not seen before, for example "`Record`", "`Pointer`", "`element`", *etc.* Some of these, such as "`Pointer`", are made visible by the `#include` lines, while others, such as "`element`", are made visible by declaring values to belong to particular types.
   The names which have meanings in `libaldor` are detailed in the documentation for that library.
3. While `OutputType` and `MiniListType(S)` are both type categories, the types in each category have different characteristics. The difference lies in what exports their types must provide:
   - Every type which is a `OutputType` must supply an output function ("`<<`").
     Since `S` is declared to be a `OutputType`, the implementation of `MiniList` can use the "`<<`" from `S` in the definitions of `MiniList`' own operations.
   - Every type which is a `MiniListType(S)` must supply several other operations, such as a constructor function called "`bracket`" to form new values, a test function called "`empty?`", and so on. `MiniList(S)` provides a `MiniListType(S)`, so it must supply all these operations. Users of `MiniList(S)` will be able to rely on these operations being available.
4. The first line of the "`add`" expression defines a type "`Rep`" (specific to "`MiniList`"). This is how values of the type being defined are really represented. The fact that they are represented this way is not visible outside the "`add`" expression.
5. Several of the functions defined in the body have parameters or results declared to be of type "`%`". In any "`add`" expression, the name "`%`" refers to the type under construction. For now, "`%`" can be thought of as a shorthand for "`MiniList(S)`".
6. There are several uses of the operations "`per`" and "`rep`" in this program. These are conversions which allow a data value to be regarded in is public guise, as a member of "`%`", or by its private representation as a member of "`Rep`".
   - `rep:  % -> Rep`
   - `per:  Rep -> %`

   These can be remembered by the types they produce: "`rep`" produces a value in `Rep`, the *rep*resentation, and "`per`" produces a value in `%`, *per*cent.
7. Some of the function definitions are preceded by the word "`local`".

This means they will be private to the "add", and consequently will not be available to users of MiniList.

8. Some of the definitions have left-hand sides with an unusual syntax:

```
[t: Tuple S]: % == ...
[g: Generator S]: % == ...
(out: TextWriter) << (l: %): TextWriter == ...
```

In general, the left-hand sides of function definitions in Aldor look like function calls with added type declarations. Some names have infix syntax, for instance "<<" above. These are nevertheless just names and, aside from the grouping, behave in exactly the same way as other names. The special syntactic properties of names may be avoided by enclosing them in parentheses. Other special syntactic forms are really just a nice way to write certain function calls. The form "[a,b,c]" is competely equivalent to the call "bracket(a,b,c)".

With this explanation, we see that the defining forms above are equivalent to the following, more orthodox forms:

```
bracket(t: Tuple S): % == ...
bracket(g: Generator S): % == ...
(<<)(out: TextWriter, l: %): TextWriter == ...
```

9. The use of the type "Generator S" is explained fully in chapter 9, so for now we will only provide a brief overview.

The function "generator" illustrates how a type can define its own *traversal method*, which allows the new type to decide how its component values should be obtained, say for use in a loop. Such a definition utilises the function "generate", in conjuction with "yield": each time a "yield" is encountered, "generate" makes the given value available to the caller and then suspends itself. This technique is described more fully in Section 9.3. When the next value is needed, the generator resumes where it left off.

Since MiniList(S) implements a "generator" function for objects of type MiniList, it is possible to iterate over them in a "for" loop. For example, in the output function "<<", we see the line

```
for s in rest l repeat out << ", " << s;
```

Here "rest l" is traversed by the generator to obtain values for "s".

2.5 Continuing ...

These first examples already give a fairly good start at reading Aldor programs. The reader is now equipped to understand most Aldor programs, at least in broad terms. For those who wish to understand the language in more detail, Part II presents further aspects of the language and revisits what we have already seen in more depth.

At this point, readers may decide to continue by studying the examples given in chapter 21, or by trying examples of their own.

PART II

The Aldor programming language

CHAPTER 3

Language orientation

3.1 Traditional and non-traditional aspects

In many regards, Aldor is a very conventional programming language:

- The language uses *explicit evaluation*, with the usual control flow. Functions play an important role, and most of a program's text consists of function definitions.
- Computed values may be saved in named *variables* or named *constants*.
- The language is *statically typed* so the nature of each variable is understood before evaluation begins. It is possible to write programs with all names explicitly declared, but the types and scopes of names are inferred if desired.
- *Overloading* of names is possible: there may be several different meanings for a given name visible simultaneously. The meaning of each occurrence is determined by the type required in context. Ambiguities may be explicitly resolved by stating the type or origin of the given name.
- Functions may be called with *named parameters*, which may have *default values*.

It is possible to transliterate most programs written in languages such as C, C++, Fortran or Lisp almost directly into Aldor.

In other regards, Aldor is somewhat unconventional:

- *Functions are first class values* and the language provides useful ways to create and manipulate them dynamically. (Many languages allow functions to be used as values, but have no way to create or combine them during execution.)
- *Types are first class values* and the language provides useful ways to create and manipulate them dynamically.

- *Dependent types* provide the freedom to defer decisions until program execution, but provide enough discipline to allow safe, efficient programs.
- Programs may make independent, *post facto extensions* to libraries. This allows existing types to belong to newly defined families without modifying the original code.
- Generators and type-specific tests provide *control abstraction*, *i.e.* a way for looping and branching expressions to use new types. While the language is statically typed, types themselves are dynamic values.
- The efficiency of *symbolic* and of *numeric* computation received equal consideration in the design of the language.

Some of these language aspects, when taken together and handled uniformly, have particularly powerful interactions.

We give one example: Since types are first class values, functions may take types as parameters and return types. Let us call such a type-to-type function a "functor." Since functions are first class, so are these functors. Therefore it is possible to write programs which manipulate functors. A functor-manipulating program is discussed in Section 21.9.

Programs such as these may be used to reorganise levels in type towers — for instance, to make the dynamic choice to convert values in the type

```
Array List Integer
```

to values in the type

```
List Array Integer.
```

3.2 Expressions and evaluation

An Aldor program consists of a set of expressions representing a computation to be performed. Performing the computation is called *evaluation* or *execution* and produces a set of *values*.

Each value has an associated *type*, which dictates how the value is to be represented in computer memory. Expression evaluation may cause other actions beyond producing values. These additional actions are called *side-effects* and include things such as output and modifying the values stored in computer memory.

The syntax of the language allows larger expressions to be built up from smaller ones. The evaluation of an expression may involve the evaluation of some or all of its subexpressions, and its value can be formed using the values of the subexpressions.

Elements such as *if*s, loops, and function forms are all expressions in Aldor. This contrasts with some other languages, such as C or Fortran, where they are special "statements" which can only appear in limited contexts.

The rules of the Aldor programming language ensure that, in a well-formed program, all the values potentially produced by any given expression have the same type. This has two consequences: it allows programs to be transformed to faster, equivalent code, by avoiding type tests during execution, and, more importantly, it allows a common class of programming errors to be detected immediately.

We say the Aldor programming language is *expression based* and *statically typed*.

3.3 Functions

Functions are treated as values in the same way as integers, lists or floating point numbers. The language provides mechanisms for composing and manipulating functions in useful ways, incorporating them in other data structures, or returning them as results.

Functions may depend on values defined externally to them so the act of dynamically creating a new function value captures the creation environment, forming what is normally called a *closure*.

3.4 Domains

Often, several functions operate within a common environment. For this reason, in Aldor, an environment is called a *domain of computation*, or *domain* for short. Environments are essentially dictionaries which tie names to values. The language allows environments to be created and manipulated dynamically, and these form the basis for abstract types, packages, and objects as first-class values.

The language provides general mechanisms to allow new types to be used in all the same ways as built-in types: they may provide the same sort of literal constants, participate in the same control structures, admit the same optimisations, *etc.* To ensure this equal status, the built-in types make use of the same general mechanisms to provide their function.

This has had two consequences: first, the extension mechanisms are pervasive and powerful; second, the language itself has very little built in. The language provides a minimal set of primitive types and operations. These are combined and extended in standard libraries to provide a rich set of facilities.

3.5 Compilation

Normally, evaluation does not directly use the source form of the expressions in a program. Rather, the evaluation is usually effected by first translating the source expressions to an equivalent set of lower-level instructions more suitable for the target environment. This is the job of the Aldor compiler. The result of the translation is a program which produces the same values and side-effects as the original program, but which might otherwise be represented very differently.

3.6 Libraries

Aldor is geared to developing and using combinations of libraries. Certain properties of the language have therefore been oriented to controlling the dependencies and interactions among libraries of programs.

A *source program* can see only the declared public behaviour of the files it uses.

A *compiled program* can depend on the private behaviour of the files it uses, but only when given explicit permission to do so. It is up to the client to decide whether it is willing to become dependent on the private behaviour of the provider.

In practice, this permission amounts to increased scope for optimisation. In no case does it allow a source-level dependency.

CHAPTER 4

Basic syntax

4.1 Syntax components

An Aldor program consists of a series of lines of text. These lines of text may be stored in a single file, or gathered from several files, or typed in as interactive input.

Some lines are not part of the Aldor program proper, but instead control its composition and the environment in which it is handled. These lines are called *system commands*. A system command is a line which has a hash character "`#`" as its *first* character. (Note that no white space may precede the "`#`" on the line.)

The example programs in this chapter use the following system commands:

```
#include "filename.as"
#pile
#endpile
```

The system command `#include "filename.as"` causes the lines of text from "`filename.as`" to be inserted into the Aldor program in place of the `#include` command.

The system commands `#pile` and `#endpile` are used to enclose lines of text in which indentation is used to determine the nesting of sequences of Aldor expressions. (See Section 4.8.)

A complete list of system commands is given in Section 22.1. System commands used in the interactive interpreter are described in Section 18.2.

When the series of lines comprising an Aldor program has been collected together, these lines are interpreted as a series of words, or *tokens*. There are several classes of tokens, each of which has a different meaning:

Identifiers such as "`Fred`" and "`rgf32`" are used as names for variables and constants.

Literals such as `42`, `1.414` and `"Urania riphaeus"`, represent explicit values. Literals are described in Section 5.2 on page 38.

Keywords such as "`if`" and "`==`" each have a special meaning in the language, and impose a special structure on neighbouring expressions.

Operators such as "`by`" and "`+`" have special syntactic properties, but are otherwise the same as identifiers (*i.e.* they are used as names for variables and constants).

Comments are used to insert free-form text into a program. A comment begins with the two characters "`--`" and continues until the end of the current line of text.

Descriptions are used to provide user-level documentation for functions, domains and categories defined in the program. A description begins with the two characters "`++`" and continues until the end of the current line of text.

White space consists of spaces, tabs, and newlines. White space is used to determine source position (line and column) information for message reporting, and for piling (See Section 4.8).

The exact rules for the syntax of each of these token classes is given in Section 22.2.3.

4.2 Escape character

The underscore character "`_`" is used as an *escape character* in Aldor to modify the interpretation of the characters which follow. For example, an escape character followed by any amount of white space (spaces, tabs, and newlines) causes the white space to be ignored, allowing the characters on either side of the white space to form a single token, such as a name or a literal.

Section 5.1 describes how the escape character can be used inside an identifier, and Section 5.2 describes how the escape character can be used inside a literal.

4.3 Keywords

The basic components of any Aldor program can be separated into two broad categories: those which are defined by the language, and those which may be defined or redefined by the program. For example, the meaning of the word "`if`" is defined by the language, and all "`if`" statements behave according to the same rules. On the other hand, the meaning of a name such as "`a`" or "`9`" or "`+`" is determined by the program in which it is used.

A *keyword* in Aldor is a word whose meaning is fixed by the definition of the language. The following words are keywords which may not be redefined:

```
add      and      always   assert   break
but      catch    default  define   delay
do       else     except   export   extend
fix      for      fluid    free     from
generate goto     has      if       import
in       inline   is       isnt     iterate
let      local    macro    never    not
of       or       pretend  ref      repeat
return   rule     select   then     throw
to       try      where    while    with
yield
```

```
.   ,   ;   :   ::  :*  $   @
|   =>  +-> :=  ==  ==> '
[   ]   {   }   (   )
```

Generally, language-defined aspects of keywords offer protocols which allow them to work with new types as well as with language-defined types. So, for example, the language-defined "`if`", provides a way for the condition to be an expression which evaluates to any type, provided that type has certain properties.

The following keywords are meaningless in the current language definition, but are reserved for future language extensions.

```
delay    fix      is       isnt      let      rule

(|  |)   [|  |]   {|  |}   `    &    ||
```

4.4 Names: identifiers and operators

A *name* is an identifier used to denote a variable or a constant. Most names begin with a letter or the character "`%`" and are made up of letters, digits and the characters "`%`", "`?`" and "`!`". The words "`0`" and "`1`" are also treated as names in Aldor so that mathematical structures can export identity elements without having to support integer literals. (See Section 5.2.)

Examples:

```
mylist     Integer     empty?     set!     %5
```

Any character may be included in a name by preceding it with the escape character ("`_`"):

```
_*PACKAGE_*   _42skidoo    mod_+      _*_+
```

When used in an identifier, the escape character is not included in the name of the identifier. To include a single underscore character in the name of an identifier, the sequence “`__`” must be used. So the name of the identifier denoted by “`mod_+`” is “`mod+`”, and the name of the identifier denoted by “`My__Integer`” is “`My_Integer`”.

A sequence of letters which would otherwise be considered a keyword (such as “`if`”) can be treated as a name by escaping one of its constituent letters (as in “`_if`”).

Certain names are treated as having special syntax properties by the language. The following identifiers can be used as infix operators, prefix operators, or both:

```
by      case    mod     quo     rem
#       +       -       +-      ~       ^
*       **      ..      =       ~=      ^=
/       /\      <       <=      <<      <-
\       \/      >       >=      >>      ->
```

Aside from their syntactic properties, these names behave just as other identifiers. See Section 4.6 for examples of using infix operators in different contexts.

A few naming conventions are used in the standard libraries:

- Names beginning with capital letters are used for types or type-producing functions.
- Names ending with a question mark are used for Boolean values, or functions which return Boolean values.
- Names ending with an exclamation mark are used for functions whose primary purpose is to perform a side-effecting (in particular, a so-called “destructive”) operation.

Note that these are only notational conventions and are not considered as part of the language.

4.5 Comments and descriptions

Comments and description strings annotate a program to help other people and other programs understand it.

A *comment* begins with the two characters “`--`” and continues until the end of the current line of text. Comments can be used to describe how a program operates, including an explanation of special assumptions made by the program, or a step-by-step description of the implementation of the algorithms used by the program. Comments are not examined by the compiler, and do not affect the meaning of a program.

A *description* begins with two or three plus characters (“`++`” or “`+++`”) and continues until the end of the current line of text. A description should be used to describe the external characteristics of a program, such as the parameters it will accept or the method used to compute the result.

Description strings are saved in the compiler output in a form accessible by other programs. If a description begins with three plus characters ("+++"), then the name it describes should appear immediately after the description. If a description begins with only two plus characters ("++"), then the name it describes should appear immediately before the description:

```
+++ An approximation to Euler's constant,
+++ which is defined as the value of the limit
+++
+++   lim(n->infinity) (1 + 1/2 + 1/3 + ... + 1/n - ln n)
+++
gamma: DoubleFloat == 0.57721_56649_01532_86060_65121;

+++ `pi' is the ratio of a circle's circumference to its diameter.
pi: DoubleFloat == 3.14159_26535_89793_23846_26434;
    ++ This is not 22/7.

Avogadro: DoubleFloat == 6.022e23;
    ++ The ratio between grams and molecular weights.
```

Both "+++" and "++" are used so that after a semicolon we can still associate a description with the previous declaration.

It is easy to remember the difference between comments and descriptions: the Aldor compiler keeps positive remarks, and throws away negative ones.

Example:

```
-- This is a quick-and-dirty move generator, with two of
-- the utility functions also made visible.

ChessPiece: with {
        bestMoves: (Board, %) -> MoveTemplate;
                ++ `bestMoves p' suggests the best moves in the
                ++ given position.

        legalMoves: (Board, %) -> MoveTemplate;
                ++ `legalMoves p' generates quasi-legal moves.
                ++ It does not handle en passant or castling.

        value: (Board, %) -> DoubleFloat;
                ++ This is a score which estimates the current
                ++ value in the given position.
}
== add {
        ...
}
```

4.6 Application syntax

Applications are typically used to denote function calls, array indexing, or element accessors for compound data types.

A prefix application typically has the following form:

```
f(a1, ..., an)
```

There are two additional forms for specifying a prefix application to one argument: juxtaposition and an infix dot.

```
f a
f.a
```

The second of these forms is completely equivalent to `f(a)`; the first is equivalent in a free-standing occurrence but associates differently — to the right, rather than the left:

```
f a b c           -- is equivalent to (f (a (b c)))
f.a.b.c           -- is equivalent to (((f.a).b).c)
f(a)(b)(c)        -- is equivalent to ((f(a))(b))(c)
```

Any application in which the argument is enclosed in parentheses ("`( )`") or square brackets ("`[ ]`") is treated as being of the "typical" form, and so associates to the left — even if a space follows the applied object. Thus "`first [1,2,3]`" is treated as identical with "`first([1,2,3])`".

The interpretation of mixed forms is determined by precedence rules: the precedence of juxtaposition is lower than that of the other forms, which are all equivalent, so an expression such as "`f(a).2(b)(c).x y`" is associated as "`((((((f(a)).2)(b))(c)).x) y)`". (A complete table of Aldor operator precedence appears in Section 4.7.1.)

Infix operators are applied to a pair of arguments using infix notation for function application:

```
a + b             -- infix notation for a call to '(+)(a, b)'
```

Infix operators are *generic* in that they can be given definitions in Aldor programs just as other identifiers. The typical form for an *infix function definition* is as follows:

```
(s1: S1) op (s2: S2) : T == E
```

where "`op`" is one of the infix operators listed in Section 4.4.

An infix operator can be used in any context where other names can be used. However, in some contexts the infix operator must be enclosed in parentheses to suppress its normal syntactic properties:

```
-- Here is a declaration for '*'.
*: (%, %) -> %

-- Here '*' is used as a variable.
* := myMultiplicationMethod

-- Here is a typical use of '*' as an infix operator.
a * b

-- Here '*' is used as a prefix operator
-- by enclosing it in parentheses.
(*)(a, b)

-- Here '*' is passed as an argument to a function.
reduce(*, 1)

-- Here '*' is being used as an argument of another infix operator.
f := (*) + g(*, 1)
```

An infix operator must be enclosed in parentheses to be used as a prefix operator. Also, an infix operator cannot appear as an argument of another infix operator unless it it enclosed in parentheses.

Alternatively, the same name may be given as an *identifier*, rather than an infixed operator, using the escape character to include special characters, for example: `_*(a, b)`.

4.7 Grouping

Complex expressions in Aldor are formed according to the precedence of the operators appearing in the expression. When an expression is formed, the operators with higher precedence form the subexpressions for the operators with lower precedence.

Parentheses ("`( )`") and braces ("{ }") can be used to override the natural precedence order defined by the language.

Because comma has a lower precedence than most other syntactic forms, it is often necessary to enclose comma-separated expressions in parentheses. We write "`f(1, 2)`", since "`f 1, 2`" would be associated the same way as "`(f 1), 2`".

Likewise, we write "`(a, b) := (1, 2)`" (see Section 5.8 for an explanation of this notation), since "`a, b := 1, 2`" would be associated as "`a, (b := 1), 2`".

Similarly, the expression "`(1 + 2) * 3`" evaluates to `9`, while the expression "`1 + 2 * 3`" evaluates to `7`, since the "`*`" operator has a higher precedence than the "`+`" operator.

Braces are normally used to enclose sequences of expressions (see Section 5.9):

```
foo(x: Integer, y: Integer): Integer == {
        a := x * y;
        b := a * a;
```

```
            b
    }
```

The meaning of an expression is the same whether braces or parentheses are used. Braces are normally used to enclose a longer expression (especially sequences) split over several lines. Parentheses are normally used to enclose shorter expressions (especially multiple values–see Section 5.8) as part of other expressions.

An implicit semicolon is assumed, if possible, after a closing brace. This is determined by whether the following token may start a new expression. For instance, in the construct

```
f: with {...} == add ...
```

introduced in Section 7.9, the "`==`" may not start an expression, so no semicolon is assumed.

To make the use of braces as natural as possible, an expression in braces may not be used as an argument to an infixed operator, *e.g.* "`+`", "`-`", "`..`". This is because many infixed operators may also be used in prefixed position. (Some, incidentally, may also be postfixed.) With infixed operators, parentheses may be used to achieve the desired effect — for example:

```
 {a; b; c}   + d     -- not ok
({a; b; c})  + d     -- ok
( a; b; c )  + d     -- ok
```

4.7.1 Precedence

Figure 4.1 provides a table of keywords and operators, given in order of syntactic precedence. Expressions are represented by "e" and keywords or operators by "$\circ$".

Each of the numbered entries in the table lists syntactic elements with the same precedence. The entries at the top of the table group most loosely, and those at the bottom most strongly. So, for example, since "`+`" is above "`*`", the expression

```
a * b + c * d
```

groups as "`(a*b) + (c*d)`". Entries with the same level number have the same precedence. For instance, "`and`" and "`/\`" have the same grouping strength.

Some operators have both unary and binary forms. Some of these operators have meanings defined in the standard base libraries (*e.g.* infixed

"`+`" and "`*`"), while others do not (*e.g.* prefixed "`=`" and "`+-`"). A programmer may provide new meanings for operators, but not for keywords. Entries for operators are flagged with "†"; keyword entries are unflagged.

This table can serve as a convenient reference for determining relative strengths of keywords and operators. The full details of the language syntax are given in chapter 22.

		Keyword/Operator				Associativity	Unary
1.		`;`				$(e \circ e) \circ e$	—
2.		`default`	`define`	`export`	`extend`	—	—
		`fluid`	`free`	`import`	`inline`	—	—
		`local`	`macro`			—	—
3.		`,`				$(e \circ e) \circ e$	—
4.		`where`				$(e \circ e) \circ e$	—
5.		`:=`	`==`	`==>`	`+->`	$e \circ (e \circ e)$	—
6.		`break`	`do`	`generate`	`goto`	—	—
		`if`	`iterate`	`never`	`repeat`	—	—
		`return`	`yield`			—	—
		`=>`				$e \circ (e \circ e)$	—
7.		`for`	`while`			—	—
8.		`and`	`or`			$(e \circ e) \circ e$	—
	†	`/\`	`\/`			$(e \circ e) \circ e$	—
9.	†	`=  ~=  ^=  >=  >  >>  <=  <  <<`				$(e \circ e) \circ e$	$\circ\, e$
	†	`case`	`is`	`isnt`		$(e \circ e) \circ e$	$\circ\, e$
		`has`				$(e \circ e) \circ e$	—
10.	†	`..  by`				$(e \circ e) \circ e$	$e\, \circ$
11.	†	`+`	`-`	`+-`		$(e \circ e) \circ e$	$\circ\, e$
12.	†	`mod`	`quo`	`rem`		$(e \circ e) \circ e$	—
13.	†	`*`	`/`	`\`		$(e \circ e) \circ e$	—
14.	†	`**`	`^`			$e \circ (e \circ e)$	—
15.		`::`	`@`	`pretend`		$(e \circ e) \circ e$	—
16.	†	`->`	`<-`			$e \circ (e \circ e)$	—
17.		`$`				$e \circ (e \circ e)$	—
18.		`add`	`with`			$(e \circ e) \circ e$	$\circ\, e$
19.		`per ref rep not  ~  #`				—	$\circ\, e$
		$A\ B$ (juxtaposition)				$e \circ (e \circ e)$	—
20.		$A(B)$ $A[B]$ $A.B$				$(e \circ e) \circ e$	—

Figure 4.1: Keyword and operator precedence

4.8 Piles

Programmers often use indentation to make the visual structure of a program conform to its syntactic structure, so that programs are easier to read. In Aldor, white space is usually ignored except to delimit tokens and to compute source position information. However, the compiler can be instructed to use indentation as part of the syntax of the language using a scheme known as *piling*.

Two system commands are used to instruct the compiler to enable and disable piling syntax, as desired, at various points in an Aldor program. The system command "`#pile`" instructs the compiler to use piling syntax for the source lines which follow, and the system command "`#endpile`" instructs the compiler to ignore initial white space on the source lines which follow.

Although the system commands "`#pile`" and "`#endpile`" are usually found in pairs, the "`#endpile`" system command can be omitted at the end of a file.

When piling syntax is being used, indentation is treated roughly as follows (see Section 22.3 for full details); a further example of an Aldor program which uses piling syntax can be found at page 209.

Expressions which are indented by the same amount are grouped together as a sequence (see Section 5.9) as though they were enclosed in braces and separated by semicolons. A sequence of expressions indented by the same amount is called a *pile*.

An expression which is too large to fit on one line at the current indentation level can be continued on another line by indenting the continuation line more than the initial line.

The indentation rules are applied first to the most indented lines, working outward to the lines which are indented the least.

The following example shows the piling rules being applied to a program which uses piling syntax, to convert it to an equivalent program which does not use piling syntax:

```
#pile
GetUp()
if Saturday then
    CookBreakfast()
    Eat << Toast << Tomato << Bacon
        << Eggs
HaveShower()
DrinkCoffee()
```

Because the line "`<< Eggs`" is indented with respect to the previous line, the two are joined.

```
#pile
GetUp()
if Saturday then
    CookBreakfast()
```

```
        Eat << Toast << Tomato << Bacon << Eggs
    HaveShower()
    DrinkCoffee()
```

The “`CookBreakfast`” and “`Eat...`” lines form a pile, which can be rewritten as a semicolon separated sequence:

```
    #pile
    GetUp()
    if Saturday then {
        CookBreakfast();
        Eat << Toast << Tomato << Bacon << Eggs
    }
    HaveShower()
    DrinkCoffee()
```

And finally the entire program is treated as a pile:

```
{
    GetUp();
    if Saturday then {
        CookBreakfast();
        Eat << Toast << Tomato << Bacon << Eggs
    }
    HaveShower();
    DrinkCoffee()
}
```

Readers wishing to experiment interactively with our examples (by using “`aldor -gloop`”) should note that piling is the default in interactive use. The examples generally should still run correctly if the illustrated layouts are used — a few may require the addition of braces (“{ }”). See chapter 18 for further details.

CHAPTER 5

Expressions

Aldor is an *expression-based* language: every construct in the language produces zero, one or more values. This chapter describes the structure of expressions in Aldor and the rules for expression evaluation.

5.1 Names

The evaluation of a name in Aldor causes the value of a variable or constant to be retrieved.

If a name refers to a variable or constant defined in the same scope or in an enclosing scope then retrieving its value is a very inexpensive operation. If a name refers to an imported constant, then there may be a cost associated with the look up, depending on the degree of optimisation used to compile the program.

Consider the following example:

```
#include "aldor"

f(n: MachineInteger): MachineInteger ==
        if n < 2 then
                1
        else
                n * f(n-1);
```

In the last line, the name "`n`" is defined in the current scope and the name "`f`" is defined in an enclosing scope, so their use is very inexpensive. The names "`*`", "`-`" and "`1`" are imported from `MachineInteger`. If this program is compiled with optimisation, then there is no cost in using the values from `MachineInteger`. Without optimisation, these values would have to be dynamically retrieved from `MachineInteger` at a modest cost.

A name may represent a *variable*, which may take on different values at different points in a computation, or a *constant*, which always refers to the same value every time it is used.

Names for constants may be *overloaded* in Aldor. That is, it is possible to have more than one constant with a given name, visible at the same point in a program. Names for variables cannot be overloaded. A name cannot represent both a variable and a constant in the same scope.

5.2 Literals

Literal constants are expressions which represent the explicit data values which appear in a program. There are three styles of literal constants in Aldor: quoted strings, integers and floating-point numbers:

```
"Aloha!"                      -- quoted string literal
10203040  16rFFFFC010         -- integer literals
1.234e56                      -- floating-point literal
```

The meaning of a constant in Aldor is determined by the environment in which it is used. For example, the constant "`1.234e56`" might be a value of type `SingleFloat` or `DoubleFloat`, depending on the context.

When a literal expression is encountered in a program, it is treated as an application of a corresponding "literal accepting" operation:

```
string:  Literal -> T
integer: Literal -> T
float:   Literal -> T
```

(where "`T`" represents the type of the value being formed). Each of these operations takes a single argument of type `Literal` and constructs a value of the appropriate type. When programs are compiled against the base Aldor library, the constant-folding optimisation will immediately convert constants of the following types to their machine representations: `String`, `MachineInteger`, `Integer`, `SingleFloat`, `DoubleFloat`.

New types may provide their own interpretation of literal constants by exporting a literal forming operation with the corresponding signature. As a consequence, if operations for creating string literals, for example, are available from several types, it may be necessary to provide a declaration to indicate which kind of literal is intended. When implementing literal-forming operations for new types, it is often useful to use the `string` literal-forming operation from `String`.

Some types may accept some literal values and not others. For example, a fixed-precision integer type might reject values which lie outside the range of values representable by the type.

String literals

String-style literals are enclosed in quotation marks. Inside a string-style literal, an underscore character "_" is used as an escape character to modify the meaning of the characters which follow:

- A quotation mark after an underscore includes a quotation mark in the literal instead of ending the literal.
- An underscore after an underscore includes a single underscore in the literal.
- White space (any number of blanks, tabs or newlines) following an underscore is ignored, and not included in the literal.

Examples:

```
"This literal is a _"string-style_" literal."
"This literal contains a single underscore: '__'."

s := "This literal is moderately long, and is broken _
     over two lines, even though the result is a single line."
```

When using `libaldor` the type `String` provides string-style literals.

Integer literals

Integer-style literals provide a syntax for whole numbers. These are in base ten unless otherwise indicated.

An integer-style literal is either

- a sequence of one or more digits, [0-9]+, with the exception of a single "0" or single "1", or
- a sequence of one or more digits giving a radix (base), followed by the letter "r", followed by a number of radix-digits using digits and/or capital letters: [0-9]+r[0-9A-Z]+.

In Aldor the numerals "0" and "1" are *not* literal constants – they are treated as names so that various mathematical structures which export 0 or 1 can do so, without being required to support general integer constants.

An underscore appearing in the middle of an integer-style literal is ignored together with any following white space. An underscore which appears at the very beginning of a word causes the whole word to be treated as an identifier, rather than as a literal constant.

Examples:

```
22_394_547

383192384712394871239482 37_
  19238749123471 2398478188_
  13982371298371 2938712391

_33    -- This word is an identifier, not an integer-style literal.

2r01010101010101010101    -- Base 2
16rDEADBEEF               -- Base 16
```

When using `libaldor`, the types `MachineInteger` and `Integer` provide integer-style literals.

Floating-point literals

Float-style literals are numbers with a decimal point, an exponent, or both.

Examples:

```
3.0       3.       3e1     6.022E+23
0.2       .2       2e-1    4.8481E-6
```

An underscore appearing in the middle of a float-style literal is ignored together with any following white space.

```
3.14159_26535_89793_23846_26433_83279_50288_41971_
  69399_37510_58209_74944_59230_78164_06286_20899_
  86280_34825_34211_70679_82148_08651_32823_06647
```

When using `libaldor`, the types `SingleFloat` and `DoubleFloat` provide float-style literals.

5.3 Definitions

A *constant* in Aldor denotes a particular value which cannot be changed. The general syntax for a constant definition is:

```
x : T == E
```

`x` is an identifier giving the name of the constant.

`T` is an expression giving the type of the constant. The type declaration is optional. If the type is declared, then the type of `E` must satisfy it. Otherwise the type is inferred from the type of the expression `E`.

`E` is an expression which computes the value of the constant.

Examples:

```
a : Integer == 23;
b == "Hello world!";
```

A function definition is a special case of a constant definition. Function definitions are described in more detail in Section 6.1.

Once a value has been given to a named constant, it cannot be changed to refer to another value.

```
-- A constant cannot be changed to refer to a different value.
num: Integer == 3;
num: Integer == 4;  -- invalid

-- In fact, it cannot be changed to refer to the same value!
hi:  String  == "'Ello!";
hi:  String  == "'Ello!"; -- invalid
```

5.4 Assignments

A *variable* in Aldor denotes a value which may change during the evaluation of a program. A variable is given a value by an *assignment* of the form:

```
x : T := E
```

`x` is an identifier giving the name of the variable.

`T` is an expression giving the type of the variable. The type declaration is optional. If the type is declared, then the type of `E` must satisfy it. Otherwise the type is inferred from the type of the expression `E`.

`E` is an expression which computes the value of the variable.

Several variables may be assigned a value at the same time:

```
(x1 : T1, ..., xn: Tn) := E
```

Any or all of the type declarations may be omitted, in which case the ith variable would read "`xi`", rather than "`xi: Ti`", and the type of `xi` is inferred from the type of the expression `E`.

Examples:

```
n: Integer := 3;
k := 3*n + 1;
n := k quo 2;

(a, b) := (1, 3);
(s: String, x) := ("Natascia", false);
```

The value of an assignment expression is the same as the value of `E`.

A special form of assignment expression is used to provide a general kind of updating operation:

```
x(a1, ..., an) := E
```

Typically, `x` is an expression which evaluates to a structured data value, such as an array or a list, and the expressions `ai` taken together specify some component of `x`. An assignment expression of this form is treated as an application of the operation "`set!`" of the form:

```
set!(x, a1, ..., an, E)
```

For example, for lists, the "`set!`" function takes as its second (component specifying) parameter a "`MachineInteger`" specifying the position of the element in the list, so we could have:

```
#include "aldor";
import from MachineInteger, List MachineInteger;
L := [1,2,3];
...
L(1) := 4;
```

which would result in `L` having the value `list(4, 2, 3)`.

The value of this form is the return value of the function "`set!`".

5.5 Functions

Function expressions are the primitive form for building functions in Aldor. An example of a function expression is:

```
(n: Integer, m: Integer): Integer +-> n * m + 1
```

See Section 6.5 for a complete description.

5.6 Function calls

Typical expressions consist mostly of function calls. For example, consider the expression

```
l: List Integer := [2 + 3, 4 - 5, 6 * 7, 8 ^ 9]
```

This has six explicit function calls (*i.e.* to the arithmetic functions "`+`", "`-`", "`*`", "`^`", to the n-ary function "`bracket`" (called using the syntax "`[...]`"), and to the type constructor function "`List`". This expression also has eight implicit function calls to the literal forming operation "`integer`".

See Section 4.6 for a complete description of the syntax of function calls.

5.7 Imperatives

The `do` expression evaluates `E` and discards the computed value, so that the `do` expression returns no value.

```
do E
```

5.8 Multiple values

A series of comma-separated expressions is used in Aldor to produce *multiple values*. The expressions are evaluated one by one, and the results are taken together as the (multiple) value of the whole expression:

```
3, 4, 5
```

This expression produces three values, all of type Integer. In general, an expression in Aldor produces zero, one or more values, each having its own type. For convenience, nevertheless, we often speak of *the* value and *the* type of an expression, even if it produces multiple values, when the intended meaning is clear from the context.

See Section 4.7 for a discussion of the use of parentheses in comma expressions.

Functions may be declared to accept or return multiple values. The example below shows how to declare, define and use a function which involves multiple values.

```
#include "aldor"
#include "aldorio"

-- Declaring a function to accept and return multiple values.
local f: (Integer, Integer) -> (Integer, Integer);

-- Defining a function to accept and return multiple values.
f(i: Integer, j: Integer): (Integer, Integer) == (i+j, i-j);

-- Using a function which accepts and returns multiple values.
(q: Integer, r: Integer) := divide f(100, 93);

-- Printing the result
stdout << "The quotient is  " << q << newline;
stdout << "The remainder is " << r << newline;
```

The call to `f` returns `(193, 7)`, which are passed as arguments to the function "`divide`" from `Integer`. This returns

```
(193 quo 7, 193 rem 7)
```

which are assigned to `q` and `r` respectively.

Comma-separated expressions are not necessarily evaluated in any particular order; furthermore, the evaluation of their subexpression may be interleaved. Thus, the program:

```
#include "aldor"
#include "aldorio"

pr2(a: MachineInteger, b: MachineInteger): () ==
    stdout << a << " " << b << newline;

n: MachineInteger := 1;

pr2({n := n + 1; n}, {n := n + 1; n})
```

could print any of "`2 3`", "`3 2`", "`2 2`" or "`3 3`", depending on the implementation and whether the code is running in a multiprocessor environment. Programs which depend on the order of evaluation of expressions to be used as arguments to a function should use a sequence to make the order explicit. (See Section 5.9.)

5.9 Sequences

A series of expressions to be evaluated one after another is called a *sequence*. A sequence is written as a semicolon-separated series of subexpressions:

```
a := 1; b := a + a; c := 3*b
```

The expressions (that is, the subexpressions of the sequence) are evaluated one by one, in the order of their occurrence, and the value of the last expression evaluated is used as the value of the sequence. A sequence may also contain one or more exit expressions, as described in Section 5.10, which prevent the evaluation of any expressions later in the sequence and so provide a way to return a value other than that of the last expression in the sequence.

Because semicolon has a relatively low precedence, it is usually necessary to enclose a sequence in braces ("`{ }`") or parentheses ("`( )`") to get the desired result. (See Section 4.7 for details.)

Examples:

```
#include "aldor"

import from MachineInteger;

n1 := (a := 1; b := a + a;  3 * b);
n2 := {a := 1; b := a + a;  3 * b}

f(i0: MachineInteger): MachineInteger == {
        a := i0;
        b := a + a;
        3 * b
}
```

The meaning of a sequence is the same whether braces or parentheses are used. Braces are normally used, especially to enclose a longer expression split over several lines. Parentheses are occasionally used to enclose shorter sequences as part of other expressions. An implicit semicolon is assumed after a closing brace but *not* after a closing parenthesis.

It is also possible to use indentation to construct sequences by enclosing lines between the directives "#pile" and "#endpile". In this context, a group of consecutive lines indented by the same amount is called a *pile* and is treated as a sequence. The precise rules for forming piles are described in Section 22.3.

5.10 Exits

An *exit expression* has the form:

```
condition => E
```

When an exit expression appears as one of the elements of a sequence, the condition is evaluated. If the condition evaluates to `true` then the value of the sequence is the value of the expression `E`, and no further components of the sequences are evaluated. If the condition evaluates to `false` then evaluation continues with the next expression in the sequence. So the expression:

```
{ a; b; c => d; e; f }
```

is equivalent to

```
{ a; b; if c then d else { e; f } }
```

In a sequence which contains an exit expression, the type of the expression `E` must be compatible with the type of the sequence, that is, with the type of the final element of the sequence[1]. If a sequence contains several exit expressions, the types of the possible exit values must all be compatible.

If the condition is not of type `Boolean`, then an implicit application of the function "`test`" is performed to convert the condition to the type `Boolean`. (See Section 12.2.)

An exit expression transfers control; it does not, itself, produce a value. As a result, the type of the exit expression is `()`, and so an exit expression can only be used in a context which does not require a value. Examples:

```
#include "aldor"

import from Integer;

b: Integer := 1;

a := { n := b * b; n < 10 => 0; n > 100 => 100 ; n }

-- 'a' will be assigned the value '0', corresponding to the
-- right hand side of the first '=>'
```

[1]In the current release of Aldor, this means that the types must be the same.

Note that all of the exit values (*i.e.* "`0`" and "`100`") have type `Integer`, and the final element of the sequence also has type `Integer`.

A series of exit expressions is often a compact way to enumerate a list of alternative cases:

```
#include "aldor"
#include "aldorio"

import from Integer;

power(base: Integer, exp: Integer): Integer == {
    exp = 0 => 1;
    exp = 1 => base;
    exp = 2 => base * base;
    exp = 3 => base * base * base;

    val := 1;

    for i in 1..exp repeat val := val * base;

    val;
}

stdout << power(2, 0) << newline;  -- Print ‘1’
stdout << power(2, 1) << newline;  -- Print ‘2’
stdout << power(2, 2) << newline;  -- Print ‘4’
stdout << power(2, 3) << newline;  -- Print ‘8’
stdout << power(2, 9) << newline;  -- Print ‘512’
```

Since any expression can appear after the "`=>`", another sequence, which may contain other exit expressions, can appear there:

```
#include "aldor"

import from Integer;

+++ If ‘a’ or ‘b’ is zero, then return 0;
+++ Otherwise return 1 if (a * b > 0), -1 elsewhere.
productSign(a: Integer, b: Integer): Integer == {
    a = 0 => 0;
    b = 0 => 0;
    a < 0 => { b < 0 => 1; -1 }
    b > 0 => 1;
    -1
}
```

If an exit expression appears as a strict subexpression of an expression other than a sequence, the exit expression is treated as a sequence of length one:

```
if b < 0 then a > 0 => flag := false;
```

This example is treated as equivalent to:

```
if b < 0 then { a > 0 => flag := false }
```

5.11 If

Conditional branching in Aldor is provided by the *if expression*.

```
if condition then T
if condition then T else E
```

When an `if` expression is evaluated, the `condition` is evaluated. If the condition evaluates to `true` then the value of the `if` expression is the value of the expression `T`. If the condition evaluates to `false` and the `else` clause is present, then the value of the `if` expression is the value of the expression `E`. If the `else` clause is not present, then the `if` expression returns no value.

An `if` expression used in a context which requires a value must have both a `then` and an `else` branch. The types of the two branches must be compatible, and the type of the expression is the type which is satisfied by both branches.

In contexts which do not require a value, the types of the `then` and `else` branch are independent. In this case, the type of the `if` expression is taken to be the type of the empty expression, and it is possible to omit the `else` branch altogether.

If `condition` is not of type `Boolean`, then an implicit application of the function "`test`" is performed to convert `condition` to type `Boolean`. (See Section 12.2)

Example:

```
#include "aldor"
import from MachineInteger;

foo(a: MachineInteger): MachineInteger ==
    if a > 0 then a * a else 0
```

Note that if the `else` clause is not present, then the `if` expression cannot be used in a context which requires a value:

```
-- This assignment is not type correct.
a : MachineInteger := if true then 1;
```

5.12 Select

A select statement takes the form:

```
select E in {
        V1 => E1;
        V2 => E2;
        ...
        En;
}
```

where `E`, `E1`, `E2` and `En` are arbitrary expressions of the same type. `V1`, `V2` are also arbitrary expressions any do not have to be the same type.

The statement is interpreted as follows: for each option (`V1`, `V2`, ...) in turn , the function `case` is called with the value of `E` as its first argument, and the value of the option as the second argument. If the call to `case` is true, then the code on the right of the `=>` is executed. If no option is satisfied, then the default code `En` is executed.

A `select` statement is type-correct if there exists `case` functions with type `(`*typeof*`(E),` *typeof*`(V)) -> Boolean` for each `V` in (`V1`, `V2`, ...).

Given an appropriate `case` function, one can write:

```
select gcd(p, q) in {
        0 => stdout << "gcd is zero";
        1 => stdout << "gcd is one";
        stdout << "gcd is complicated"
}
```

```
select getOption() in {
           0 => option1();
           1 => option2();
        2..5 => optionX();
        illegalOption();
}
```

Unions also export a case function, so one can do:

```
U ==> Union(var: Symbol, poly: Poly);
u: U := ...
select u in {
        var => ...
        poly => ...
        never
}
```

5.13 Logical expressions

Logical expressions in Aldor are provided using the following forms:

```
E1 and E2
E1 or  E2
not E
```

An "and" expression is true if both E1 and E2 are true. If the first expression evaluates to false, then the second expression is not evaluated.

An "or" expression is true if E1 or E2 or both are true. If the first expression evaluates to true, then the second expression is not evaluated.

A "not" expression is true if E is false.

If any of E1, E2 or E is not of type Boolean, then an implicit application of the function "test" is performed to convert the value to type Boolean. (See Section 12.2) The type of a logical expression is Boolean.

Examples:

```
#include "aldor"
#include "aldorio"

if true and false then stdout << "This string will not be printed.";

import from MachineInteger;

-- Define ‘test’ for MachineInteger:  a null value returns false.
test(x: MachineInteger): Boolean == if x = 0 then false else true;

if 1 or 1 then stdout << "I can do this." << newline;

-- Define ‘test’ for String:  an empty string returns false.
-- Note that ‘#s’ (‘the length of s’) uses
-- the previous test for MachineInteger.
test(s: String): Boolean == if #s then true else false;

if true and 1 and "I can do this, too." then
    stdout << "This string will be printed." << newline;
```

The logical connectives "and", "or" and "not" are syntactic elements of the language that should not be confused with similar functions exported from the type Boolean (/\, \/ and ~). The functional versions from Boolean evaluate all of their arguments before computing their result, and denote function values. The logical connectives cannot be used as functions:

```
#include "aldor"
#include "aldorio"

import from List Boolean;

-- This expression will print:  ‘[T,T,F]’
stdout << ((map ~) [false, false, true]) << newline;
```

but you cannot say: ((map not) [false, false, true]), since "not" is *not* a function. The Aldor function "map", defined for all *BoundedFiniteLinearStructureType* types. In the case of Lists, "map" is defined as a function taking a function (for example, " ") as an argument and returning a function taking a "List" as an argument and returning a "List".

5.14 Loops

Repeat

The Aldor language provides a set of constructs to handle loops in a way which is both elegant and efficient. The concept lying at the base of Aldor loops is that of *generator*, discussed in depth in chapter 9. Generators provide an abstract way to traverse values belonging to certain domains without violating the principle of *encapsulation* (see Section 7.8, page 88). Generators can be considered as autonomous entities producing values. This section shows how values they produce can be used to control loops.

The general form of a loop expression is:

Iterators `repeat` *Body*

The iterators control how many times the body is evaluated. Any number of iterators may be used on a loop, and each is either a "`while`" or a "`for`" iterator.

A loop with no iterator repeats forever, unless terminated by an error or some other event. For example:

```
#include "aldor"
#include "aldorio"

-- This will repeat forever...
repeat {
        stdout << "Row, row, row your boat," << newline;
        stdout << "Gently down the stream."  << newline;
        stdout << "Merrily, merrily, merrily, merrily," << newline;
        stdout << "Life is but a dream."     << newline;
        stdout << newline;
}
```

Two forms, "`break`" and "`iterate,`" may be used within the loop body to control the loop evaluation, as described later on pages 55 and 56.

While-iterators

A "`while`" iterator allows a loop to continue so long as a condition remains true. The syntax of a "`while`" iterator is

`while` *condition*

If a loop has a "`while`" iterator, then at the beginning of each iteration *condition* is evaluated. The result is then tested:

- If it is *true*, the evaluation of the loop *proceeds*.
- If it is *false*, then the loop is *terminated*.

Just as with "`if`" and other expressions having tests for control flow, if the condition of a "`while`" is not a `Boolean` value, then an appropriate "`test`" function is applied to determine the sense of the condition.

The following example shows a `repeat` loop using a `while` iterator:

```
#include "aldor"
```

```
#include "aldorio"

import from Integer;

n := 10000;
k := 0;         -- ‘k’ counts the number of iterations.

while n ~= 1 repeat {
        k := k + 1;

        if odd? n then n := 3*n + 1 else n := n quo 2;
}

stdout << "Terminated after " << k << " iterations." << newline;
```

This loop counts the number of times the body has to be evaluated in order for `n` to reach one. When `n` reaches 1, the evaluation of the loop is terminated, and the count is printed. (It is a well-known conjecture that this process will terminate for all integers $n > 0$. This is sometimes called the "$3n + 1$ problem".)

In a case such as this it is important to give `k` an initial value, since the loop may be iterated zero times! In fact, if the `while` condition is false the first time that it is evaluated, then the `repeat` body will be not executed at all:

```
#include "aldor"
#include "aldorio"

import from Integer;

n := 0;

while n > 0 repeat {
    -- This will never be executed.
    stdout << "Hello world!" << newline;
}
```

For-iterators

Very often, loops are used to traverse certain kind of data structures, such as lists, arrays or tables, or to execute some operations for all the numerical values in a defined range.

"`for`" iterators make this sort of loop convenient to write. Take, for instance, the following examples:

```
#include "aldor"
#include "aldorio"

-- Add up the elements of a list, return the sum.
sum(ls: List Integer): Integer == {
        n := 0;
        for i in ls repeat n := n + i;
        n
}

-- Add up the elements of an array, return the sum.
sum(arr: Array Integer): Integer == {
```

```
        n := 0;
        for i in arr repeat n := n + i;
        n
}

-- Add up the odd numbers in a range, return the sum.
sum(lo: Integer, hi: Integer): Integer == {
        n := 0;
        if even? lo then lo := lo + 1;

        for i in lo..hi by 2 repeat n := n + i;
        n
}


import from List Integer, Integer;

-- Use ‘sum: List Integer -> Integer’:
stdout << sum([1,2,3,4])     -- Prints ‘10’.
```

The "`for`" iterators in these examples all have the form

`for` *lhs* `in` *Expr*

The *lhs* part specifies a variable which will take on the successive values over the course of the iteration. The *lhs* has the form

[`free`] *name* [: *Type*]

The type declaration is optional. If it is missing, the type of the variable is inferred from the source of the values, *Expr*.

The "`free`" part of the *lhs* determines the scope of the variable. Without it, a new variable of the given name is made local to the loop. If the word "`free`" is present, then the loop uses an existing variable from the context. In this case, the value of the variable is available after the loop terminates.

The example on page 50 can be rewritten using a free loop variable:

```
#include "aldor"
#include "aldorio"

import from Integer;

n := 10000;
k := 0;                      -- Make a top-level variable ‘k’.

for free k in 1.. repeat {   -- Use above ‘k’ freely in the loop.

        if odd? n then n := 3*n + 1 else n := n quo 2;

        if n = 1 then break;  -- exit the loop if n = 1
}

-- Here the last value of ‘k’ is available.
stdout << "Terminated after " << k << " iterations." << newline;
```

The previous example uses the "`break`" construct explained later in this section, on page 55. When a `break` is evaluated it causes the termination of the loop. In this case, the `break` is the only exit point of the loop, because we are iterating over an open segment, producing infinitely many values: 1, 2, 3, ...

Now we turn our attention to the expression traversed by the "`for`" iterator. In the examples, we have seen

- a list, "`ls`",
- an array, "`arr`", and
- integer segments, "`lo..hi by 2`" and "`1..`".

In general, the "`for`" iterator expression must have type `Generator` *T*, where T is the type of the "`for`" variable. If the expression is not of this type, then an implicit call to "`generator`" is inserted. Any visible `generator` function of an appropriate type will be used. See chapter 9 for a description of generators and how to create new ones.

The examples we have seen work because the list, array and segment types provided by `libaldor` export `generator` functions.

There is a second form of "`for`" iterator which filters the values used. This has the form:

```
for lhs in Expr | condition
```

This kind of "`for`" iterator skips those values which do not satisfy the condition. For example, the `sum` example we saw earlier could have been written as:

```
#include "aldor"

-- Add up the odd numbers in a range
sum(lo: Integer, hi: Integer): Integer == {
        n := 0;
        for i in lo..hi | odd? i repeat n := n + i;
        n
}
```

Multiple iterators

A "`repeat`" loop may have any number of iterators, with the following syntax:

iterator$_1$ ... *iterator*$_n$ `repeat` *Body*

The loop is repeated until one of the iterators terminates it: the first "`while`" which has a false condition or the first "`for`" which consumes all its values ends the loop.

This is convenient when the termination condition does not relate directly to a "`for`" variable, or when structures are to be traversed in parallel.

Continuing the example on page 52, we may use a "`while`" to decide if the end has been reached, and, at the same time, use a "`for`" to count the number of times we have evaluated the loop body:

```
#include "aldor"
#include "aldorio"

import from Integer;

n := 10000;
k := 0;

while n ~= 1 for free k in 1.. repeat
        if odd? n then n := 3*n + 1 else n := n quo 2;

stdout << "Terminated after " << k << " iterations." << newline;
```

Multiple "`for`" iterators can allow lists to be combined in an efficient way:

```
#include "aldor"
#include "aldorio"

import from List Integer, Integer;

l1 := [1,2,3];
l2 := [8,7,6,5];

-- Add the pair-wise products in two lists.
x := 0;
for n1 in l1 for n2 in l2 repeat
        x := x + n1 * n2;

stdout << "The result is " << x << newline
```

These two "`for`" iterators are used in parallel, like a zipper combining the two lists. The loop stops at the end of the shortest list, in this case giving a sum of three products.

This is not a double loop – to use *all* pairs with the number from `l1` and the second number from `l2`, you would use two nested loops, each with its own "`repeat`":

```
#include "aldor"
#include "aldorio"

import from List Integer, Integer;

l1 := [1,2,3];
l2 := [8,7,6,5];

x := 0;
for n1 in l1 repeat
        for n2 in l2 repeat
                x := x + n1 * n2;

stdout << "The result is " << x << newline
```

Using more than one iterator is often the most efficient natural way to write a loop. A loop with two "`for`" iterators is more efficient than

```
...
m: MachineInteger := min(#l1, #l2);
for i in 1..m repeat
        x := x + l1.i * l2.i
```

because it does not need to traverse the lists each time to pick off the desired elements. And the code is more concise than

```
...
t1 := l1;
t2 := l2;
while (not empty? t1 and not empty? t2) repeat {
        x := x + first t1 * first t2;
        t1 := rest t1;
        t2 := rest t2;
}
```

Break

Evaluating a "`break`" causes a loop to terminate. For example:

```
#include "aldor"
#include "aldorio"

import from Integer;

n := 10000;
k := 0;         -- 'k' counts the number of iterations.
```

```
repeat {
        if odd? n then n := 3*n + 1 else n := n quo 2;

        k := k + 1;

        if n = 1 then break;
}

stdout << "Terminated after " << k << " iterations." << newline;
```

"`break`" is most useful when the condition which quits the loop falls most naturally in the middle or at the end of the loop body. Sometimes it is possible to change a test which appears at the end to a test at the beginning. This helps make programs more readable, since one sees immediately what will cause the loop to end. Also, it allows the exit to be controlled by a "`while`" iterator, rather than an "`if`" and a "`break`".

If a "`break`" occurs inside nested loops, it terminates the deepest one.

Iterate

Evaluating an "`iterate`" abandons the current evaluation of the loop body and starts the next iteration. For example:

```
#include "aldor"
#include "aldorio"

import from Integer;

n := 10000;
k := 0;         -- 'k' counts the number of iterations.

repeat {
        k := k + 1;
        if odd? n then { n := 3*n + 1; iterate }

        n := n quo 2;
        if n = 1 then break;
}

stdout << "Terminated after " << k << " iterations." << newline;
```

This example does the same thing as the previous one, but is organised slightly differently. The "`iterate`" on the second line of the loop body causes the rest of the body to be skipped.

"`iterate`" can be used instead of placing the remainder of a loop body inside an "`if`" expression. This can make programs easier to read, by emphasising that certain conditions are not expected, and by avoiding extra levels of indentation. It is particularly useful when the decision to go on to the next iteration of a loop is buried deep inside some other logic, rather than appearing at the top level of the loop body.

An "`iterate`" is equivalent to a "`goto`" branching to the end of the loop body. Thus, the meaning of "`iterate`" is independent of whether there are any "`while`" or "`for`" iterators controlling the loop.

If an "`iterate`" occurs inside nested loops, it steps the deepest one.

Definition in low-level terms

It is possible to express the loop behaviour in terms of `goto`s and labels (see section 5.17 for details). A loop of the form

it_1 it_2 ... it_n `repeat` *Body*

is equivalent to

{
  $init_1$; $init_2$; ... $init_n$;
  `@TOP`
  $step_1$; $step_2$; ... $step_n$;
  Body;
  `goto TOP;`
  `@DONE`
}

Where, if it_i is "`while` $cond_i$", then $init_i$ is empty and $step_i$ is

`if not` $cond_i$ `then goto DONE`

if it_i is "`for` lhs_i `in` $expr_i \mid cond_i$", then $init_i$ is

g_i `:= generator` $expr_i$`;`

and $step_i$ is

`step!` g_i`;`
`if empty?` g_i `then goto DONE`
lhs_i `:= value` g_i
`if not` $cond_i$ `then goto TOP`

(a "`for`" without a condition is treated as if it had the condition "`true`").

In this, `TOP`, `DONE` and the g_i are generated names, and are not accessible to the other parts of the program.

5.15 Generate expressions

Generate expressions are used to create generators. For a complete description of how to use `generate` to create a generator, see chapter 9. The general syntax for a generate expression is:

```
generate E
generate to n of E
```

`E` is an expression which represents code which will be evaluated each time a value is extracted from the generator. Evaluation begins at the start of the expression `E` and continues until an expression of the form

```
yield v
```

is processed, where `v` is the value to be passed back to the context which is collecting values from the generator. Each time a value is requested after the first value is yielded, control resumes after the `yield` expression which produced the previous value.

5.16 Collections

A *collect* expression provides a convenient shorthand for creating generators and aggregate objects. A collect expression has the form:

$E \ iter_1 \ \ldots \ iter_n$

where $n \geq 1$. The $iter_i$ are iterators, as described in Section 5.14. Consider the following example:

```
#include "aldor"
#include "aldorio"
...
import from Integer, List Integer;
stdout << [x*x for x in 1..10] << newline;
stdout << [x*y for x in 1..10 for y in 10..1 by -1] << newline;
```

This program creates a list of the squares of the integers from 1 to 10, and then a list of products of integers. Note that `while` can form an iterator, and can therefore be used in a collect expression.

Note that the square brackets are not part of the collect expression, but are simply a shorthand for a call to the function "`bracket`", with the value of the collection as an argument. The domain "`List Integer`" from the Aldor base library exports a function with the signature `bracket: Generator Integer -> %`, which is called twice in the above example.

As collect expressions produce generators, one would expect that generate expressions and collect expressions are related. The collect expression:

$E \ iter_1 \ \ldots \ iter_n$

is equivalent to the generate expression:

`generate` $iter_1 \ \ldots \ iter_n$ `repeat yield` E

Thus, `x*y for x in 1..9 for y in 9..1 by -1` is the same as:

`generate for x in 1..9 for y in 9..1 by -1 repeat yield x*y`

Collect expressions provide a convenient notation for creating new aggregates, and require no additional functionality in the language.

5.17 General branching

Aldor provides unconditional branching using *label expressions* and *goto expressions*. A label expression is of the following form:

```
@L E
```

where `L` is an identifier known as the *label name*, and `E` is any expression. The type of the label expression is the same as the type of `E`. Names which are used as labels have no relationship with variables or constants of the same name, and a label name may also be used as a variable or constant. A label name obeys the same scope rules as constants or variables, but may not appear in any expressions other than `goto`s and other label expressions. Since labels are constants, it is an error to bind the same label twice in the same scope.

A goto expression has the following form:

```
goto L
```

where `L` is the name of a label. After the evaluation of a goto expression, execution resumes with the expression associated with the label `L`. The type of the goto expression itself is the type `Exit`.

The label must appear in the same function body as the `goto`. In addition, it is an error for a `goto` to branch into an inner scope of the scope in which it appears (that is, a local function or any repeat or collection including a `for`, `where`, `add` or `with` expression). A `goto` may also not branch out of a function or a `with` or `add` expression.

Example:

```
foo(a: MachineInteger): MachineInteger == {
    if a <= 0 then goto ERROR;

    return a * a;

    @ERROR
    stdout << "The argument must be a positive value!" << newline;
    0
}
```

If the first test is successful, then the "`return`" expression is skipped and the execution proceeds on the line following "`ERROR`".

Labels can also be defined at the top level of a file, since the top level of a file is treated as a sequence:

```
#include "aldor"
#include "aldorio"

@LAB1
stdout << "You will see this forever..." << newline;
goto LAB1;
```

5.18 Never

The expression "`never`" is a special value, of type `Exit`, which acts as a programmer-supplied assertion that execution will never reach that point. An exit expression can be useful as it may allow a program to be translated into more efficient code.

The following Aldor code is a possible use of `never`

```
s := {
        x = 0 => "zero";
        x > 0 => "positive";
        x < 0 => "negative";
        -- This expression is unreachable
        never
}
```

With luck, the "`never`" at the end of the sequence will not be reached in any execution of the program. (If it is reached, Aldor will complain "`Aldor error:  Reached a "never"`").

CHAPTER 6

Functions

Functions lie at the heart of Aldor: typical expressions consist mostly of function calls.

Much of what is done by *ad hoc* means in other languages is done in Aldor through normal functions. It is the job of the compiler to ensure that relying on functions in this way does not adversely affect performance.

This chapter describes how to define and use functions, beginning with typical examples of function definition and application, then describing more specialized features, including keyword arguments, default arguments, function expressions (also called "anonymous functions"), and curried functions.

6.1 Function definition

A typical function definition has the following form:

```
f (s1: S1, ..., sn: Sn) : T == E
```

This definition has a number of parts:

- the *function name*, `f`
- the *formal parameter names*, `s1,...,sn`
- the *formal parameter types*, `S1,...,Sn`
- the *return type*, `T`, and
- the *function body*, `E`.

The *function name* is an identifier which will be used to denote the function. In a given scope there may be more than one functions with the same name; in this case the function name is said to be *overloaded*.

The *formal parameter names* are identifiers which are used to refer to the values passed to the function as arguments. The formal parameter names are visible in the body of the function, in the types of the formal parameters, and in the return type (See Section 8.12).

The *formal parameter types* are type-valued expressions (*e.g.* "`Integer`" or "`SquareMatrix(n+m, Complex Float)`") which specify what type of value is expected as the corresponding actual argument to the function.

The *return type* is a type-valued expression which specifies the type of the value computed by the function.

The *function body* is an expression which, when evaluated, produces the return value of the function. The type of the value returned by the function body must be compatible with the given return type.

More elaborate forms of function definitions are described in sections 6.4 and 6.6.

Multiple return values

Just as functions can take any number of parameters, they may also return any number of results. The typical function definition given above is a special case of the more general form:

```
f (s1: S1, ..., sn: Sn) : (t1: T1, ..., tm: Tm) == E
```

Now in place of a single return type we have:

- the *return names*, `t1,...,tm`, and
- the *return types*, `T1,...,Tm`.

This function definition takes n arguments and returns m results.

The *return names* are identifiers which can be used to refer to the values returned from the function. The return names are visible in the types of the return values.

The *return types* are type-valued expressions which specify the type of the corresponding value returned from the function.

Any or all of the return names may be omitted, in which case the ith return declaration would read "`Ti`", rather than "`ti: Ti`".

When a function has no formal parameter or no return value an empty pair of parentheses is used as the formal parameter list or return value list. For example, the following function takes no parameter and returns no result:

```
f () : () == E
```

Return expressions

Inside the body of a function definition, a *return expression* is used to explicitly pass control back to the calling environment and to return values from the function. The general form of the return expression is:

```
return E
```

The value of the expression “E” is returned as the value of the function. The type of “E” must be compatible with the declared return type of the function. The return expression itself has type “Exit”.

Inside a function which returns more than one value the return expression may explicitly supply more than one value:

```
return (v1, ..., vn)
```

Inside a function which returns no value, an empty return expression may be used:

```
return
```

Since the value of the function body is used as the value of the function, in many cases an explicit return expression is unnecessary:

```
f (n: Integer) : Integer == if n < 1 then 1 else n * f(n-1)
```

The value returned by the function “f” is the value returned by the “if” expression. An explicit “return” is not needed.

6.2 Function application

A typical (prefix) function application has the following form:

```
f (a1, ..., an)
```

This form of function application has the following parts:

- the *function*, f, and
- the *actual parameters*, a1,...,an.

The *function* is an expression which denotes the function to be called. In general, “f” can be any expression whose value is a function.

The *actual parameters* are expressions which specify the values to be passed as arguments to the function. The types of the actual parameters must be compatible with the type of the function “f”.

As discussed in Section 5.8, the order of evaluation of the actual arguments in an application is not defined. More elaborate forms of function application are developed in sections 6.3, 6.5 and 6.6.

When a function takes no parameter, an application of that function must have as its actual parameters an expression which produces no value. Often, such an application takes the following form:

```
f ()
```

Other application notations

In addition to the normal prefix application notation there are a small number of special syntactic forms in Aldor denoting function application. The general reason behind these rules is to make programs more readable. For example there is a set of infix operators (see Section 4.4), so that you can write "`a + b`" instead of "`+(a,b)`". Furthermore, some language forms cause implicit application of functions:

- the treatment of literal values,
- the application of one object to another,
- the updating of objects,
- the interpretation of tests in conditional statements
- the mechanism for generating a set of values for iteration.

Refer to chapter 12 for a more detailed description of these forms.

6.3 Keyword arguments

Consider the following function definition:

```
-- Compute a point on the line with slope 'm' and intercept 'b'.
line(x: DoubleFloat, m: DoubleFloat, b: DoubleFloat): DoubleFloat ==
    m*x + b;
```

Because all the parameters have the same type, it may be difficult to remember which one is which. As a result, the meaning of a call such as

```
line(3.2, 8.2, 1.0);
```

might not be readily apparent.

One way to increase the readability of such a program is to place the arguments in named variables before calling the function:

```
slope     := 8.2;
intercept := 1.0;

line(3.2, slope, intercept);
```

But this approach needlessly increases the number of variables used by the program. In addition, now the values for the slope and intercept are not explicitly visible at the call point. So one sort of unreadability has been exchanged for another (and *remembering* the order of the parameters is no easier than before).

Keyword arguments

An alternative in Aldor is to allow the actual arguments in an application to be supplied by name using *keyword arguments*. For example:

```
line(3.2, b == 1.0, m == 8.2);
```

An actual argument in this form of application has the following parts:

- the *formal parameter name*, (*e.g.* `b`),
- the double-equal symbol "`==`" and
- the *actual parameter value*, (*e.g.* `1.0`).

The *formal parameter name* is an identifier which must match one of the formal parameter names given in the definition of the function.

The *actual parameter value* is then used as the value of the formal parameter with the same name, regardless of its position in the actual argument list in the function application. The type of the actual parameter value must match the type of the formal parameter with the same name.

Any parameters supplied as keyword arguments must appear after any arguments supplied by position alone. It is an error if any of the formal parameters is not supplied with a value, either as a positional argument or by using a keyword argument.

6.4 Default arguments

In a programming language where function names may not be overloaded, such as Lisp or C, some functions are written to take a variable number of arguments. When these functions are called, they decide which arguments they have been passed and what to do about the missing ones.

In a language such as Aldor, where function names may be overloaded, there are often several functions visible with the same name. Which function to use is decided on the basis of the number of actual arguments supplied and their types, and possibly on the type of the return value required by the context of the function application.

So, instead of writing functions which take a variable number of arguments, in Aldor we are allowed to write several functions with the same name, each with a fixed number of arguments. One advantage of doing this is that the decision on which function to call can be made once, at compile time, whereas code in the body of the function, to supply missing arguments, would be exercised in each time the function was run.

So, continuing the example from Section 6.3, we can write:

```
-- Compute a point on the line with slope 'm' and intercept 'b'.
line(x:DoubleFloat, m:DoubleFloat, b:DoubleFloat):DoubleFloat ==
    m*x + b;

-- Assume a default intercept of '0'.
line(x:DoubleFloat, m:DoubleFloat):DoubleFloat == line(x, m, 0);
```

Soon afterward, however, we want to give other arguments default values. Then the number of functions needed increases exponentially in the number of arguments which are to have default values.

Another problem arises when arguments have the same type: it is not always possible to overload the function name enough to provide each argument with a default value.

For these reasons, languages with name overloading sometimes provide an explicit way to supply default values to named parameters.

Default arguments

In Aldor, default values for named parameters can be supplied in the definition of a function using the following form:

```
f (s1: S1 == v1, ..., sn: Sn == vn) : T == E
```

This form of function definition has the following additional part:

- the *default argument value specifications*, in which "`==`" (the double-equal symbol) introduces each of the *default argument values*, `v1,...,vn`.

The default argument values are expressions which, when evaluated, produce values, each of which can be used as the corresponding argument to the function. The type of the default value for a parameter must be compatible with the corresponding parameter type.

Any or all of the default argument values may be omitted, in which case the form of the ith formal parameter would read "`si: Si`" instead of "`si: Si == vi`". A function definition which supplies a default argument value for one of its parameters must also supply a default argument value for each of the following parameters.

Once again continuing the example given above, we can now define a single function which allows any combination of its final two arguments to be omitted:

```
-- Compute a point on the line with slope 'm' and intercept 'b'.
-- The default slope is 1, and the default intercept is 0.

DF ==> DoubleFloat;

line(x: DF, m: DF == 1, b: DF == 0) : DF == m*x + b;
```

This definition supplies a default value of "`1`" for "`m`", and a default value of "`0`" for "`b`". Some example applications of the function "`line`" are as follows:

```
x: DoubleFloat := 3.2;

stdout << line(x, 8.0) << newline;       -- 8.0 * x + 0
stdout << line(x) << newline;            -- 1   * x + 0
stdout << line(x, b == 5.0) << newline;  -- 1   * x + 5.0
```

In an application of a function whose definition supplies default argument values, it is an error if any of the formal parameters is not supplied with a value, either as a positional argument, or by using a keyword argument, or by using the default argument value supplied (if any) for that formal parameter. The default argument values are evaluated, if and when they are used in a function application, as the other actual parameters are evaluated. As discussed in Section 5.8 the order of evaluation of the actual arguments to an application is not defined.

6.5 Function expressions

Function expressions are the primitive form for building functions in Aldor. The general form for a function expression is:

```
(s1: S1 == v1, ..., sn: Sn == vn) : (t1: T1, ..., tm: Tm) +-> E
```

The syntax "`(s: S) : T +-> E(s)`" for a function expression is intended to resemble the mathematical notation $s \mapsto E(s)$ for a function specification. The infix keyword "`+->`" denotes the λ operator from a typed lambda calculus. A function expression has many of the same parts as a function definition, including the formal parameter names/types, the return names/types, the function body (see Section 6.1), and default arguments (see Section 6.4).

When a function expression is evaluated, it captures the lexical environment in which it appears, creating a lexical closure. The values of the variables which are visible in the scope of the function expression are then available when it is eventually applied to a set of arguments.

More typical cases of function expressions are formed in much the same way as the corresponding cases of function definitions. For example, the expression:

```
(f: Integer -> Integer, n: Integer) : Integer +->
    if n < 1 then 1 else n * f(n-1)
```

represents an integer function of two arguments which bears some superficial resemblance to a factorial function.

Since a function expression has the same parts as a function definition except for the name, function expressions are sometimes known as *anonymous functions*. Function expressions are also known as *lambda expressions* in many programming languages.

Function types

Like any other expression, a function expression can be assigned a type. The type assigned to a function expression is called a *function type*. A function type is formed with the infix keyword "`->`":

```
(s1: S1 == v1, ..., sn: Sn == vn) -> (t1: T1, ..., tm: Tm)
```

The syntax "`S -> T`" for a function type is intended to be reminiscent of the mathematical notation $S \to T$ for a set of functions. A function type has many of the same parts as a function definition, including the formal parameter names/types, the return names/types (see Section 6.1), and default arguments (see Section 6.4).

For function types, any or all of the formal parameter names may be omitted, in which case the ith parameter declaration would read "`Si`" rather than (in the most general case) "`si: Si == vi`".

The type of "`f`", ("`Integer -> Integer`"), in the function expression shown previously, is a typical example of a function type. Any function expression which takes one integer argument and returns one integer result is a member of this type.

The ability to include the formal parameter names and default arguments in the specification of the type is useful when using keyword arguments (see Section 6.3), default arguments (see Section 6.4), and dependent types (see Section 14.2).

Function definition revisited

The typical form for a function definition given in Section 6.1 is really just a shorthand for the following equivalent definition:

```
f : (s1: S1, ..., sn: Sn) -> T == (s1: S1, ..., sn: Sn) : T +-> E
```

This form makes it easier to see that function definitions are the same as definitions of any other values. Specifically, the expression:

```
n : Integer == 8
```

defines a type ("`Integer`") and a value ("8") for the identifier "`n`". In the same way the function definition:

```
f (n: Integer) : Integer == if n < 1 then 1 else n * f(n-1)
```

which can also be written as:

```
f : (n: Integer) -> Integer ==
    (n: Integer) : Integer +-> if n < 1 then 1 else n * f(n-1)
```

defines a function type ("`(n: Integer) -> Integer`") and a function expression for the identifier "`f`".

Function application revisited

Since function expressions evaluate to functions, they can be used in the place of the function name in a function application:

```
((n: Integer) : Integer +-> if n < 1 then 1 else n * (n-1))(5);
```

This way of calling function expressions may be rather verbose. However, function expressions can also be assigned to local variables:

```
g := (n: Integer) : Integer +-> if n < 1 then 1 else n * (n-1);
g(5);
```

So once again we see that a typical function definition is merely a special case of a more general framework, based on the fact that function expressions can be used just like any other expression.

6.6 Curried functions

Since function expressions can be used just like any other expression, we can write a function which returns a function as its result:

```
DF ==> DoubleFloat;
line (m: DF, b: DF) : (x: DF) -> DF ==
        (x: DF) : DF +-> m*x + b
```

The function "`line`" is a function which has two formal parameters, and which returns a function of one formal parameter. The type of the function returned by "`line`" is "`(x: DoubleFloat) -> DoubleFloat`".

A function which returns a function as its result is called a *curried function*, after Haskell Curry, a significant contributor to the theory behind functional programming[1].

To simplify the definition of curried functions in Aldor, two shorthand notations are provided.

Function expressions revisited

First, we generalise the syntax of function expressions (see Section 6.5) by inductively defining the *curried function expression*

```
(s1: S1) ... (sn: Sn)(s0: S0) : T +-> E
```

to be equivalent to the expression

```
(s1: S1) ... (sn: Sn) : (s0: S0) -> T +-> (s0: S0) : T +-> E
```

In the definition above, we have assumed a single formal parameter within each set of parentheses and a single return type, to simplify the exposition. Note that "`->`" and "`+->`" associate from right to left and that "`->`" groups more tightly than "`+->`".

Using the equivalent method of writing function definitions, given in Section 6.5, and applying this shorthand to the right-hand side, the definition of the function "`line`" given above can be written as:

```
DF ==> DoubleFloat;
line : (m: DF, b: DF) -> (x: DF) -> DF ==
        (m: DF, b: DF)(x: DF) : DF +-> m*x + b;
```

[1] See, for instance, *Combinatory Logic,* Curry and Feys, North Holland, Amsterdam 1958.

Note that the function "`line`" is still a function which has two formal parameters, and which returns a function of one formal parameter. The only additional notation used here is the curried function expression used to define the value of "`line`". An expression for "`line`" can be written without using a curried function expression, but the shorter form is more convenient.

Function definition revisited

As a second shorthand for defining curried functions, we define the *curried function definition*

```
f (s1: S1) ... (sn: Sn) : T == E
```

to be equivalent to the expression

```
f : (s1: S1) -> ... -> (sn: Sn) -> T ==
    (s1: S1) ... (sn: Sn) : T +-> E
```

Continuing the example from the previous paragraphs, we can now express the definition of the function "`line`" in its most convenient form:

```
DF ==> DoubleFloat;
line (m: DF, b: DF)(x: DF) : DF == m*x + b;
```

As a further example, if we define exponentiation for `MachineInteger` functions of a `MachineInteger` as follows:

```
MI ==> MachineInteger;
(f:(MI->MI))^(n:MI) : (MI->MI) == {
   n = 0 => (x:MI):MI +-> x;
   n = 1 => f;
   (x:MI):MI +-> f((f^(n-1))x);
}
```

an alternative notation could be defined, using the curried function conventions, as:

```
multApply(n:MI,f:(MI->MI)):MI->MI == f^n
```

Function application revisited

The application of curried functions to their arguments needs no additional machinery: a curried function is just a function which returns a function, and any expression (including the result of a function application) can be used as a function as long as the actual parameters to the application are compatible with the type of the function. For example:

```
#include "aldor"
#include "aldorio"
DF ==> DoubleFloat;
import from DF;

line(m: DF, b: DF)(x: DF): DF == m*x + b;

stdout << "f(x) is  x - 1" << newline;

stdout << "f(1.0) = " << line(1,-1)(1.0) << newline;
stdout << "f(2.0) = " << line(1,-1)(2.0) << newline;
stdout << "f(3.0) = " << line(1,-1)(3.0) << newline;
```

So in the application "`line(1,-1)(1.0)`", the curried function "`line`" is applied to the arguments "`1`" and "`-1`", which returns a function of one argument, which is applied to the argument "`1.0`".

As a result, we can use the "`line`" function in this example to create other functions:

```
#include "aldor"
#include "aldorio"
DF ==> DoubleFloat;
import from DF;

line(m: DF, b: DF)(x: DF): DF == m*x + b;

f := line(1, -1);
stdout << "f(x) is  x - 1" << newline;

stdout << "f(1.0) = " << f(1.0) << newline;
stdout << "f(2.0) = " << f(2.0) << newline;
stdout << "f(3.0) = " << f(3.0) << newline;
```

When we supply in this way only some of the arguments to a function, the result is a new function related to the original. This technique is a basic feature of functional programming, and is known as *currying* a function.

Some languages provide an automatic conversion of functions of type `(A, B) -> C` to functions of type `A -> B -> C`. This automatic conversion is *not* done in Aldor. When desired, this conversion can be made explicitly with a function expression:

```
#include "aldor"
#include "aldorio"
import from Integer;

I ==> Integer;

-- Curry the function '*'
ctimes == (a: I)(b: I) : I +-> a * b;
times3 == ctimes 3;

stdout << "3 * 2 = " << times3 2 << newline;

-- Convert general functions:
curry(f: (I, I) -> I)(a: I)(b: I) : I == f(a,b);

stdout << "3 + 2 = " << curry(+)(3)(2) << newline;
```

CHAPTER 7

Types

This chapter describes the world of types in Aldor.

To use Aldor effectively, it is important to understand how to make type declarations. Type declarations allow one to write the same sorts of programs that one can write in C or Fortran. The base libraries provide a rich set of types and constructors, sufficient for many purposes. The chapter begins by describing how to create simple type expressions for declarations.

To take fuller advantage of Aldor, it is useful to understand how to create new types and environments. In this area, Aldor provides considerably more power than other programming languages. The remainder of the chapter explains the language primitives for forming new environments and shows how they may be used to provide parameterised types and packages.

7.1 Why types?

Values are ultimately stored in a computer as sequences of bits. A given sequence of bits, however, can have different interpretations when used by different programs.

For example, on one computer the 32 bits

```
01000101011010000011111100000000
```

can represent

- the integer `1164459776`,
- the floating point number 3.7159375×10^3, and
- the character string `"Eh?"`.

A type provides an interpretation of binary data as a value which a program can manipulate. Different programming languages have different ways to associate types with data.

Types on Data: Some languages, such as Lisp, incorporate types in values, augmenting the representation of a value with extra bits encoding the type. In such languages, each data value is self-identifying. The implementation of the bits which encode the type can be done either in hardware or software. One goal of optimising compilers for these languages is to determine cases when it is possible to avoid storing and checking these type codes.

Types on Variables: Other languages, such as Fortran-77, associate types with variables, either via declarations or by implicit rules. Associating a type with each variable requires less optimisation to be efficient, but can be less flexible than the previous approach. Also, this method can provide a greater degree of safety, since certain mistakes can be detected before program execution.

Types on Faith: Finally, some languages, such as B, a predecessor of C, have different operations to interpret data in different ways. Any data value can be used in any operation, and it is the responsibility of the programmer to get things right. For example, one operation will interpret a set of bits as pointer and another will interpret it as an integer.

Most modern programming languages associate types with values, with variables or both.

Object-oriented programming languages tend to adopt the "**Types on Data**" approach. Variables might be declared to belong to certain classes, but objects generally carry type information in the form of object-specific methods. Some object languages, such as C++, achieve efficiency by treating primitive and non-primitive types differently. The programmer must constantly remember the difference — for example, it is not possible to derive new classes from `int` or `char *`. Other object languages uniformly pair types with data values, giving what is sometimes called "objects all the way down."

Aldor, on the other hand, adopts the "**Types on Variables**" approach, and values are not normally self-identifying. This approach allows uniform, efficient treatment of primitive and program-defined types. The flexibility that we have come to expect from object systems is obtained by promoting types to be first-class values.

Treating types as first-class values leads to a greater versatility than typical object systems. For example, programs may be parameterised by types provided at execution time. Many values may be declared as belonging to exactly the *same* execution-time type, or to types having some defined relationship. Combining operations can be made safe without requiring execution-time tests. The usual sort of object behaviour is easily recaptured. For example, self-identifying data may obtained by incorporating types in the values.

7.2 Type expressions

Expressions in Aldor may compute values which are types. For example, all of the expressions in **bold face** in the program below produce type values:

```
#include "aldor"
import from DoubleFloat;
i := Integer;
l := List Integer;
af:  Array DoubleFloat := [1.0, 2,0, 3.0];
myfun(T: PrimitiveType):  PrimitiveType == Array T;
mytype := myfun DoubleFloat;
```

In this example we see a number of expressions computing type values, some of which are assigned to variables, others passed as parameters, and still others used in declarations. Type values are typically formed by applying type-constructing functions to values of one sort or another.

Certain kinds of types in Aldor are used in specific ways.

A *domain* is a type which defines a collection of exported symbols. The symbols may denote types, constants and functions. Many domains also define an interpretation for data values, called a *representation type*; these domains are also known as *abstract data types.* Those domains which are not abstract datatypes are called *packages.*

A *category* is a type which specifies information about domains, including the specification of the public interface to a domain, which consists of a collection of declarations for those operations which may be used by clients of the domain.

The next several sections describe properties of types, which are common to all types. Section 7.8 describes those properties of types which are specific to domains, packages and abstract datatypes. Section 7.9 describes properties of categories.

The language defines a number of types and type-constructing functions. These are described in chapter 14. Most programs also use types or type constructors from various libraries. Section 15.2 describes standard libraries used with Aldor.

7.3 Type context

Whereas any expression *may* be used to compute a type value, there are certain contexts in which a type is *required*. These contexts are indicated by T in the following expressions (details of which may be found by consulting the index):

- ... : T — declaration
- ... :* T — declaration

- … `$` T — selection
- … `::` T — type conversion
- … `@` T — type constraint
- … `pretend` T — type lie
- T `with` … — primitive type former
- T `add` … — primitive type former
- `import` … `from` T — bring names into scope
- `inline` … `from` T — allow program dependency
- `export` … `from` T — export names from scope
- `export` … `to` T — export names to foreign environment

We say that the expression T is in *type context*.

Type context is special in two ways:

- Expressions occurring in type context are not guaranteed to be evaluated in any particular order, or even evaluated at all.
- Identifiers occurring in an expression, T, appearing in type context must be constant in the scope in which T occurs.

Type evaluation

The reason expressions in type context are not guaranteed to be evaluated in order, or at all, is to give maximum flexibility to produce efficient programs. A portable program will only use non-side-effecting expressions in type context.

The expressions occurring in type context should be viewed as an annotation of a program rather than as part of the computation. For instance, the same program may be written with declarations, type restrictions and selections being inferred or explicit in varying degrees. So the program

```
#include "aldor"
#include "aldorio"
import from Integer;

n := 2 + 2;

stdout << n << newline
```

has the same meaning as

```
#include "aldor"
#include "aldorio"
import from Integer;

n: Integer := 2 +$Integer 2;

stdout << n << newline
```

which has the same meaning as

```
#include "aldor"
#include "aldorio"
import from Integer;

I ==> Integer;

n: I := (+$I @ (I,I) -> I) (2@I, 2@I);

stdout << n << newline
```

Name constancy

As a related but more fundamental concern, an expression in type context must only contain names which are constant over the scope in which the type occurs, because, without this rule, it would not be possible to associate well-defined types to expressions.

Consider the following *incorrect* function, which uses a domain called `MachineIntegerMod` (see AXIOMimodnSample for a definition) which is parameterised by an `Integer`:

```
f(n: Integer): () == {
        local a, b: MachineIntegerMod(n);

        a := coerce((3 * n + 1) quo 2);
        b := coerce((5 * n + 1) quo 3);

        if n > 4 then   n := n + 1;  -- This line is ILLEGAL!

        c := a - b;

        stdout << "The result is " << c << newline
}
```

The problem is that if `n` is updated, the type of `a` and `b` is no longer valid and there is no reasonable interpretation for “`-`” or “`c`”.

The names appearing in an expression in type context may be

- defined via “`==`”,
- imported via “`import`”, or
- function parameters which are nowhere updated.

7.4 Dependent types

In many programming applications it is not possible to determine the specific type of a family of related types, that an expression will have, until it is evaluated. There are a number of possible ways to treat this situation:

Solution 1: Forget the type information.

If it is known that the result is some kind of stored structure, then pretend it is a `Pointer`.

This solution has obvious drawbacks:

- The private representation information known about the result is compromised, creating a source-code dependency which can be difficult to manage.
- The result might not really be a pointer — it might be an integer, a floating point number or something else which uses memory differently.
- Mistakes are more difficult to detect.

Solution 2: Use a Union type.

If there is a fixed set of possible result types, then one could return a Union type which includes the lot.

This solution is only acceptable when the set of types is fixed and the logic associated with each case is relatively distinct. If the different cases are treated in essentially the same way, then this solution encourages a programmer to treat the various union branches with the same code.

Solution 3: Use an Object-Oriented approach.

One can associate the dynamic type of the value with the value itself.

If the different possible types are to be handled in essentially the same way, then this solution can be a reasonable choice. The type information associated with the value can carry the functions which may be used to operate on it.

One difficulty with this solution is that it is not possible to indicate when many values belong to the *same* dynamic type, leading either to unsafe code, or to testing tags during program execution.

Solution 4: Use Dependent Types

A *dependent type* is a type T in which the *type* of one subexpression of T depends on the *value* of another.

Dependent types allow compile-time type checking for values whose types depend on other values which will only be present during program execution. This powerful tool is not often provided in other programming languages. In Aldor, it is a basic feature of the language which allows a great deal of flexibility in supporting various programming paradigms.

Dependent types

Consider the following example:

```
#include "aldor"
#include "aldorio"

sumlist(R: ArithmeticType, l: List R): R == {
        s: R := 0;
        for x in l repeat s := s + x;
        s
}

import from List Integer, Integer, List SingleFloat, SingleFloat;

stdout << sumlist(Integer,     [2,3,4,5]) << newline;
stdout << sumlist(SingleFloat, [2.0, 2.1, 2.2, 2.4]) << newline;
```

The *type* of the parameter "`l`" and the type of the result of the function each depend on the *value* of the parameter "`R`".

Because of this dependency, we say that the type of `sumlist` is a dependent type, in this case the dependent mapping type

```
(R: ArithmeticType, l: List R) -> R
```

Note that within `sumlist` it was *known* that *all* the elements of the list `l` had type `R`. The "`+`" operation does not need to make any run-time checks to verify that both of its operands are of the same type.

Dependency is a relationship among values which arises because of the occurrence of one value in the type of another. In Aldor, dependencies between values are allowed in the following contexts:

- function parameters
- function results
- cross product values
- signatures in a category `with`-expression

In addition, the function result types are allowed to depend on the values of the function parameters.

Uses and examples

Dependent types allow parametric polymorphism, as in the `sumlist` function shown above. They also allow a program to compute a result, and return with the result a context for interpreting the result.

For example, a function could be declared of type:

```
eigenValues: (R: Ring, Matrix R) ->
             (E: AlgebraicExtension R, Vector E)
```

A function satisfying the above declaration could be defined to create a new arithmetic domain in which the eigenvalues of a matrix may be defined, and a vector of eigenvalues could then be created in that domain. Both the new domain and the vector of eigenvalues could then be returned by the function. Operations for manipulating the eigenvalues can then be retrieved from `E` to operate on values from the vector.

A complete example of a dependent type constructing function has been seen already in Section 2.4. There the `MiniList` function takes a parameter `S: OutputType` and returns result of type `MiniListType S`.

An example manipulating dependent mapping values is given in Section 21.9. An example treating dependent type-value pairs as objects is given in Section 21.10.

With types as values in Aldor, dependent types allow the specification of relationships among types, which can be an extremely powerful programming tool.

Mutually dependent types

Values may have *mutually* dependent types. For example, the type constructing functions defined by the language include:

```
->: (Tuple Type, Tuple Type) -> Type
Tuple: Type -> Type
```

As another example, functions may be declared with mutually dependent arguments:

```
Ladder: (D: with {f: % -> E}, E: with {g: % -> D} ) -> Type
```

Note that these mutual dependencies are in the *type* dimension; this type of dependency is analogous to the more usual dependency between expressions in the *value* dimension:

```
RecA == Record(head: A, tail: Union(nil: Pointer, b: RecB));
RecB == Record(head: B, tail: Union(nil: Pointer, a: RecA));
```

7.5 Subtypes

Type declarations in Aldor associate properties with constants and variables. These declared names may then be used in expressions and the rules for well-formed expressions determine the properties associated with the built-up expressions.

The most important property of an expression is how to interpret the data representing the value of the expression. Every value in Aldor is a member of a *unique domain* which determines the interpretation of its data.

Sometimes it is necessary to associate additional properties with values, and to provide rules to manipulate the values on the basis of the properties they satisfy. Subtypes provide the mechanism for manipulating the additional properties associated with values.

A *subtype* is a type whose members lie in a particular domain and satisfy a particular property. The domain to which the members belong is the *base domain* of the subtype. The vocabulary of properties differs from one base domain to another.

A value in Aldor may be a member of any number of subtypes. By definition, the base domain for each of these subtypes is the same as the unique domain of the value.

We use $\mathit{Subtype}(D)$ to denote the set of all subtypes on the base domain D. Suppose T_1 and T_2 are two subtypes from $\mathit{Subtype}(D)$. If all the values belonging to T_1 also belong to T_2, then we say T_1 is a *subtype* of T_2 and write $T_1 \sqsubseteq T_2$. We also say that T_2 is a *supertype* of T_1 and write $T_2 \sqsupseteq T_1$. For any particular base domain D, the collection of subtypes $\mathit{Subtype}(D)$ forms a lattice under the relation $\sqsubseteq$. The empty subtype is the bottom element and the subtype consisting of *all* elements of D is the top element. The subtypes based on one domain can have no relationship with subtypes based on another domain.

Not all domains support a vocabulary of properties. For this first version of the language, only the domain of all domains and the domain of all maps provide properties which lead to non-trivial subtype lattices. It would be a compatible extension to the language to allow arbitrary `Boolean`-valued functions to be used as properties for subtyping purposes.

The following rules define the relation $\sqsubseteq$:

Categories: For any category-valued expression C, let $Ex(C)$ denote the set of exported symbols which must be provided by any domain $D \in C$. Then

$$C_1 \sqsubseteq C_2 \iff Ex(C_1) \supseteq Ex(C_2)$$

As a corollary of the above definition:

$$D \in C_1 \wedge C_1 \sqsubseteq C_2 \Rightarrow D \in C_2$$

Inheritance for domains from categories is analogous to class membership and inheritance between categories is analogous to class containment.
Note that Aldor is constructed so that a domain is only a member of a named category if it explicitly inherits from the category — not if it merely exports the same collection of (explicit) declarations[1]. For named categories C_1 and C_2, it is only the case that $C_1 \sqsubseteq C_2$ if C_1 inherits from C_2, either directly or indirectly.

Mapping types: The subtyping rule for mapping types is derived from the fact that a function which maps $S_1 \to T_1$ can be used in any context which provides a value of type S_1 as an argument, or a value of type S_2 where $S_2 \sqsubseteq S_1$:

$$S_1 \to T_1 \sqsubseteq S_2 \to T_2 \iff S_2 \sqsubseteq S_1 \wedge T_1 \sqsubseteq T_2$$

Additionally, in determining the relation of mapping types, an argument or return with a keyword is a subtype of one without:

$$(A_1, ..., a_i : A_i, ..., A_n) \sqsubseteq (A_1, ..., A_i, ..., A_n)$$

If $T_1 \sqsubseteq T_2$ then any value of type T_1 can be used in any context which requires a value of type T_2.

[1] In the current implementation, each named category C implicitly exports a symbol named "`%%`" with type C (see Section 8.12), whose presence is tested for in checking category membership.

7.6 Type conversion

A *type conversion* is an operation which changes a value from one type to a value of another type to which the original value would not otherwise belong. As a general rule, Aldor does not automatically convert a value from one type to another. For example, an integer is not automatically converted into a floating-point number. Such conversions must be made explicitly in the text of a program.

As discussed in Section 7.5, a value in Aldor may be viewed as a member of more than one type by virtue of the subtype relation on types. In this case no conversion is necessary.

Primitive conversions

The language provides one primitive type conversion operation: `pretend`. A "`pretend`" expression is used to lie about the type of a value.

Expr `pretend` *Type*

causes the value computed by *Expr* to be treated as though it were of type *Type*. `pretend` is the only operator in Aldor which is not type-safe: using `pretend` can lead to unchecked type errors which only reveal themselves when a program is executed. For this reason `pretend` should be used with caution. For example, one could use "`pretend`" to examine the bit-level representation of data when a type does not provide operations to do so.

Two additional type-safe operations are defined in Aldor using pretend: `rep` and `per`. These operators convert between the public and private views of a type, and are discussed in Section 7.8.

Conversion functions

Most type conversions are performed by functions. By convention, type conversion is most often performed by a "`coerce`" function, exported either by the source type or the destination type. Each library of Aldor programs may establish its own set of conventions regarding how conversion functions are named, and where they are implemented.

Examples of conversion functions could include:

- `coerce: MachineInteger -> %` from `DoubleFloat`
- `coerce: MachineInteger -> %` from `Integer`
- `coerce: SInt$Machine -> %` from `MachineInteger`
- `coerce: % -> SInt$Machine` from `MachineInteger`

Such functions are so common that a special syntax is defined to allow `coerce` functions to be called conveniently. The syntax

Expr `::` *Type*

is a shorthand for the application

`coerce(`*Expr*`) @` *Type*

(see Section 8.3).

Courtesy conversions

While most type conversions must be made explicitly, a very conservative set of *courtesy conversions* are performed as needed. Courtesy conversions change between items represented as

- multiple values,
- a single `Cross` product value (see Section 14.4) and
- a single `Tuple` value (see Section 14.3).

Certain Aldor programs would be extremely pedantic if courtesy conversions were not applied.

The following courtesy conversions are applied automatically as required:

- `Cross(T,  ..., T)  -> Tuple T`
- `Cross(T1, ..., TN) -> (T1, ..., TN)`
- `Cross(T)           -> T`

- `(T,  ..., T)  -> Tuple T`
- `(T1, ..., TN) -> Cross(T1, ..., TN)`

- `T -> Tuple T`
- `T -> Cross T`

These conversions allow functions which take or return multiple values to be used to pass arguments to other functions which can accept them, without requiring notation for an explicit conversion.

These conversions are applied only when the type of an expression exactly matches one of the conversions. For example, a value of type `List Cross(T, T, T)` would *not* automatically be converted to a value of type `List Tuple T`. Such a conversion could incur a significant hidden cost, even in more ordinary circumstances.

There is not at present any mechanism for a program to specify additional courtesy conversions.

7.7 Type satisfaction

We say that a type S *satisfies* the type T if any value of type S can be used in any context which requires a value of type T.

The following rules define the satisfaction relation among types in Aldor:

Exit: For any type T, the language-defined type `Exit` satisfies T. This rule allows an expression of type `Exit` to be used wherever any type is expected.

Category: A type S satisfies the language-defined type `Category` if S is the type of a category.

Type: A type S satisfies the language-defined type `Type` if S is the type of a domain or category. In other words, all domains and categories are types.

No value: For any type S, the type S satisfies the type `()`. In other words, any value can be used in a context where no value is expected. The values are simply thrown away, and the expression is treated as having returned no value.

Subtypes: If S is a subtype of T, then S satisfies T. See Section 7.5.

Courtesy conversions: A type S satisfies a type T if a courtesy conversion exists which converts S to T. The courtesy conversion rules are described in Section 7.6.

7.8 Domains

A *domain* is an environment providing a collection of exported constants. These may include exported types, functions or values of other sorts.

It is useful to think of domains as being of two kinds: abstract datatypes and packages. An *abstract datatype* is a domain which defines a distinguished type and a collection of related exports. A *package* is a domain which does not define a distinguished type but does provide a collection of related exports. That is, a package is a domain which is not an abstract data type. This distinction is merely a convenience, though, as one is merely a special case of the other, as will be seen below.

Creating domains with "add"

The primitive for forming domains is the `add` expression. The syntax for an `add` expression is

[Expr] `add` *AddBody*

The left-hand side *Expr* is an optional domain-valued expression, which specifies a domain from which any or all of the exports of the `add` can be inherited. *Expr* is called the *parent* (or parent domain) of the `add`.

The expression *AddBody* is typically a sequence of definitions which implement the exports of the domain.

Examples of the different kinds of domains, all of which can be created by `add` expressions, are to be found throughout the remainder of this section.

The value of the expression `A add B` is a domain which exports those symbols exported by `A` and those exports defined in `B`. The type of the expression `A add B` is

```
C with { x1: T1; ...; xn: Tn }
```

where `C` is the type of `A`, and `x1,...,xn` are the symbols defined (using `==`) in `B`, whose types are `T1,...,Tn`, respectively. Note that the types `T1,...,Tn` are allowed to contain references to the values `x1,...,xn`.

Packages

Packages are the simplest form of domains: they group a number of values together in an unordered collection that can be imported as a unit. A package formed with a single "`add`" expression can be used to provide functions operating in a common environment, and packages may be combined using binary "`add`" operations.

The following simple package provides functions for keeping score at a baseball game:

```
add {
        single(n: Integer): Integer == n;
        double(n: Integer): Integer == n + n;
        triple(n: Integer): Integer == n + n + n;
        homer (n: Integer): Integer == n + n + n + n;
}
```

The exports from a package may include values belonging to different types, including various functions among different types. Conditional exports (see below) can also appear in packages.

Units conversion example

Here is an example of a package which exports both types and functions:

```
#include "aldor"
#include "aldorio"

Lengths == add {
    Centimetres == MachineInteger;
    Inches == MachineInteger;

    import from DoubleFloat;

    local round(f: DoubleFloat): MachineInteger == {
         import from Integer; -- for 'machine'
         tmp: MachineInteger := machine truncate f;
         if ((f - tmp::DoubleFloat) > 0.5) then tmp := tmp + 1;
         tmp
    }

    inches(c:Centimetres):Inches == round(c::DoubleFloat/2.54);
    centimetres(i:Inches):Centimetres == round(i::DoubleFloat*2.54);

}

import from Lengths;

i : Inches == 1 + 1 + 1;

stdout << i << newline;

c : Centimetres == centimetres i;

stdout << c << newline;
```

This package exports two types `Inches` and `Centimetres`, with operations for converting between them.

Abstract datatypes

Often, a domain exports a distinguished type and a collection of operations on that type — in general, enough operations to make the type a sufficiently interesting object. This situation is so overwhelmingly common that additional features in the language are used to support it.

An *abstract datatype* is a domain which defines a distinguished type and a collection of constants and functions related to the type.

The definition of an abstract datatype includes a specification of the representation of values belonging to the type and the implementation of operations which manipulate those values.

Inside an add expression which denotes an abstract datatype, the domain value created by the expression is used as the unique type exported by the expression. Within the add expression, this domain is denoted by the identifier "%".

We will often use the term "domain" when speaking of a particular abstract datatype, if the meaning is clear from the context. In abstract datatype terminology, a package may be described as a degenerate abstract datatype which does not provide any exports on its type. Such a structure is sometimes called an *empty carrier*.

The following example illustrates a simple abstract datatype:

```
add {
        I ==> Integer;
        import from I;

        0 : % == 0@I pretend %;
        1 : % == 1@I pretend %;

        (x: %) = (y: %) : Boolean == x@% pretend I = y@% pretend I;

        ~ (x: %) : % == if x = 1 then 0 else 1;
        (x: %) \/ (y: %) : % == if x = 1 \/ y = 1 then 1 else 0;
        (x: %) /\ (y: %) : % == if x = 1 /\ y = 1 then 1 else 0;
}
```

This domain expression includes constant definitions for identifiers "0" and "1" which belong to the domain and a few useful function definitions.

Since % denotes a domain, it can be used in type context just like any other type. In particular, operations defined in an add expression can be disambiguated by using a package call of the form "x$%", which denotes a value named x defined in the add expression. Furthermore, inside B in a domain definition of the form:

```
D : C == A add B
```

(described below, under "Visibility of inherited operations") the type expression D is treated as equivalent to the type expression %, since the domain D is defined as the value of the add expression, which is denoted by % inside B. The fact that % and D are equivalent is not visible outside the add. The equivalence is a property of the definition. The scope of % is bound by the add.

Outside the add expression, the type expression D has no relationship to any instances of % which may be visible.

Representation

The *representation type* of the elements of a domain is the domain which describes how data values belonging to the domain are encoded as sequences of bits.

Inside an `add` expression which denotes an abstract datatype, the constant "`Rep`" is used to refer to the representation type of the elements of a domain. Since `Rep` is a domain, it can be used in type context just like any other type.

`%` and `Rep` denote two different types in Aldor. While a value of type `%` belongs to the public view of the domain, a value of type `Rep` belongs to the private view of the domain. For example, suppose that the representation type for a polynomial domain is a list domain. We would then say that a polynomial value is represented by a list: a list is not a polynomial, nor is a polynomial a list.

The operations "`rep`" and "`per`" provide a type-safe mechanism for converting data values between the public and private views of domain elements: `rep` converts a value of type `%` to the representation type `Rep`; `per` converts a value of type `Rep` to the public type `%`.

```
rep x ==> x @ % pretend Rep;
per r ==> r @ Rep pretend %;
```

An easy way to remember which conversion operation to use is to remember the assertions `rep(x)@Rep` and `per(r)@%`: the result from `rep` has type `Rep` and the result from `per` has type `%` (*per*cent).

To illustrate the use of `Rep` in a domain definition, consider the following version of the simple `add` expression given above:

```
add {
        Rep == Integer;
        import from Rep;

        0 : % == per 0;
        1 : % == per 1;

        (x: %) = (y: %) : Boolean == rep x = rep y;

        ~ (x: %) : % == if x = 1 then 0 else 1;
        (x: %) \/ (y: %) : % == if x = 1 \/ y = 1 then 1 else 0;
        (x: %) /\ (y: %) : % == if x = 1 /\ y = 1 then 1 else 0;
}
```

The representation type is defined as `Integer`, from which the domain imports the operations it needs to use. The constants `0` and `1` are defined by viewing corresponding values from the representation as values of the domain itself. Similarly, the equality operation is defined using the equality operation from the representation.

Separating a domain and its representation provides a clear line of demarcation between the public and private views of domain values. In this way, Aldor is an example of an *abstract datatype language*: the representation of a domain provides the private view of its elements which is used to define functions which operate on values from the domain. In public, values from the domain are viewed as belonging to the domain itself.

The operations provided by a domain can be arranged so that client programs which use a domain need not depend on the representation of the domain. When the representation of a domain is localised to the domain definition in this way, the representation is said to be *encapsulated* by the domain definition.

The representation of a domain is typically a more primitive domain which is chosen to achieve a certain level of efficiency in the operations provided by the domain. At the lowest representational level lie the built-in machine types provided by the Aldor base library, described in Section 14.16 and in Section 15.1.

Each of the machine types is a domain whose representation depends on the architecture of the machine being used. While the actual representations of machine-level values are not specified by the language, typically most or all of them will have efficient runtime implementations. The availability of these built-in types provides a great degree of flexibility in designing efficient representations for Aldor domains.

Domain inheritance

A domain may *inherit* the implementation of many of its operations from another domain, called its *parent*, by placing the parent domain on the left-hand side of an `add` expression.

For example, when the `add` expression:

```
Integer add {
        Rep == Integer;
        import from Rep;

        0 : % == per 0;             -- This definition is redundant.
        1 : % == per 1;             -- This definition is redundant.

        -- This definition is redundant.
        (x: %) = (y: %) : Boolean == rep x = rep y;

        ~ (x: %) : % == if x = 1 then 0 else 1;
        (x: %) \/ (y: %) : % == if x = 1 \/ y = 1 then 1 else 0;
        (x: %) /\ (y: %) : % == if x = 1 /\ y = 1 then 1 else 0;
}
```

is used to define a domain which exports the operations `0`, `1`, and `=`, these operations can be inherited from `Integer`, and so need not be implemented explicitly in the `add` expression.

A domain may inherit operations from another domain only if the representation type of the parent domain is compatible with the representation type of the domain. In many cases the representation type of the domain will be taken to be the parent domain itself. Packages and abstract data types may inherit from packages without reservation since there is no possibility of representation mismatch in this case.

Visibility of inherited operations

Operations provided by a domain are often defined using other operations provided by the domain. When some operations are inherited from a parent domain, it is important that these operations be visible in the add expression. In the domain definition[2]

```
MyBit : BooleanArithmeticType with {
        0 : %;
        1 : %;
}
== Integer add {
        ~ (x: %) : % == if x = 1 then 0 else 1;
        (x: %) \/ (y: %) : % == if x = 1 \/ y = 1 then 1 else 0;
        (x: %) /\ (y: %) : % == if x = 1 /\ y = 1 then 1 else 0;
}
```

the definitions for `~`, `\/`, and `/\` each use the operations `0`, `1`, and `=` from `MyBit`, which are inherited from `Integer`. These inherited operations are made visible in the `add` expression by the following rule: whenever an expression `A add B` appears in a context requiring a domain whose type is the category `C`, then any operations required by `C` which are not defined in `B` are taken from the domain `A`.

So in the above example, the `add` expression used to define `MyBit` appears in a context which requires a domain which satisfies `BooleanArithmeticType` and provides the exports `0` and `1`. `BooleanArithmeticType` is a category provided by the standard Aldor library, which includes, among other things, a declaration for "`=`":

```
= : (%, %) -> Boolean;
```

Since the domain `Integer` (from the base Aldor library) also implements an operation `=` with the same signature, this operation is made visible on the right-hand side of the `add`.

Note that the statement `import from Integer` would also make the `=` operation from `Integer` visible, but its signature would instead be:

```
= : (Integer, Integer) -> Boolean;
```

and so it could only be used to compare `Integer`s, not `MyBit`s.

Conditional definitions

A *conditional definition* is a definition which is only provided by a domain under certain assumptions. For example:

[2] This example uses a `with` to declare the type of the domain, which will be explained in section 7.9.

```
Zmod (n: Integer) : ArithmeticType with {
        if prime? n then inv : % -> %;
}
== Integer add {
        if prime? n then {
                inv (x: %) : % == ...
        }
}
```

Z_n, the domain of integers modulo n, is always a Ring. However, if n is prime, then Z_n is also a Field, meaning that it should provide a multiplicative inverse for nonzero values. In an `add` expression, a definition which appears in the consequence of an `if` expression is said to be a conditional definition: the domain only provides the operation if the condition in the `if` expression evaluates as true.

7.9 Categories

In Aldor a *category* is used to specify information about domains. Categories allow domain-manipulating programs to place restrictions on the sort of domains they are prepared to handle and to make promises about the domains which they may return. The restrictions and promises are expressed in terms of collections of exports which the domains in question will be required to provide.

All type values have "`Type`" as their unique base type. As with all other values, it is the unique base type which determines how values are to be represented.

On the other hand, a domain may belong to any number of categories so long as it has the necessary exports. That is, the world of categories provides a sub-typing structure on `Type`.

This section describes how to create and use categories.

Creating categories with "with"

The primitive for forming categories is the `with` expression. The syntax for a `with` expression is

[Expr] `with` *WithBody*

The left-hand side is an optional `Category`-valued expression. Allowing a nonempty expression on the left-hand side is merely a syntactic convenience, and is equivalent to placing the expression as a part of the right-hand side:

L `with` { *R* }

is equivalent to

`with` { *L* ; *R* }

The right-hand side of a `with` expression contains a specification of the set of exports which must be provided for a domain to belong to this category. The specification is typically a sequence containing any of the following types of expression:

- declarations
- category-valued expressions
- conditional expressions
- default definitions

The following is an example of a simple category:

```
with {
        scale: (Integer, Integer) -> Integer;
        n: Integer;
}
```

For a domain to satisfy this category, it must provide at least two exports: "`scale`", a function of the given type, and "`n`", an `Integer`. For example, the domain "`D`" as defined in the following program satisfies this category:

```
D == add { import from Integer; n == 3; scale == *; extra == 0 }
```

Here, the type of `D` is inferred to be

```
with{n: Integer; scale: (Integer, Integer) -> Integer; extra: Integer}
```

It is quite usual to see `with` expressions as the declared types in definitions. The definition of `D` could just as well have been written as

```
D: with {
        n:     Integer;
        scale: (Integer, Integer) -> Integer;
        extra: Integer
}
== add {
        import from Integer;
        n == 3;
        scale == *;
        extra == 0;
}
```

This form has the advantage that the interface to the domain is explicitly shown.

The exports specified by a `with` expression may have mutually dependent types. That is, it is possible to export types and operations on them. For example:

```
with {
        DecomposedMatrix: Type;

        decompose: Matrix R -> DecomposedMatrix;

        solve: (DecomposedMatrix, Vector R) -> Vector R
}
```

(where `R` is a parameter).

In general, any number of related types and operations may be given:

```
#include "aldor"

with {
        T: Type;
        L: ListType T;

        x: T
        lx: L;
}
```

Typically, a domain will export a type and enough operations on that type to make it a sufficiently interesting object. It therefore makes sense to equate the exporting domain and the type for which the exports are being defined. The name used for this unified domain is "`%`", and is implicitly exported by all types formed using "`with`" and "`add`".

Thus, the "`DecomposedMatrix`" example above may be rewritten by replacing `DecomposedMatrix` with `%`:

```
with {
        decompose: Matrix R -> %;
        solve: (%, Vector R) -> Vector R
}
```

A domain satisfying this type would have the following form:

```
DecomposedMatrix: with {
        decompose: Matrix R -> %;
        solve: (%, Vector R) -> Vector R
} == add {
        Rep == ...

        decompose(m: Matrix R): % == ...
        solve(dm: %, v: Vector R): Vector R == ...
}
```

Export from

In addition to the exports specified in the declarations of a `with` expression, a category may also *cascade* exports from other domains. That is, when a domain satisfying some category is imported, operations from other domains may be implicitly imported. The `export from` form is used to specify the additional domains to be imported.

For example,

```
with {
        decompose(m: Matrix R): % == ...
        solve(dm: %, v: Vector R): Vector R == ...

        export from Matrix R;
}
```

When a domain satisfying this category is imported, `Matrix R` will also be imported. As with the `import from` statement (see Section 8.4), it is possible to restrict the imports which are cascaded:

```
export {+: (%, %) -> %} from Matrix R;
```

This form will import only the "+" operation from `Matrix R`.

The cascaded exports of a category do not affect type satisfaction questions in any way.

Defaults

It is a common situation that, if we are given a category satisfying a number of operations, new operations can be defined on domains in that category, which only use the information supplied in the category. For example, a datatype with an equality operation may be declared as:

```
with {
        =:  (%, %) -> Boolean; ++ equality
}
```

A domain which satisfies this category is free to decide its own implementation of equality. As it stands, this is fine but we may want further operations on the datatype, such as a not-equals operation, `~=`.

```
with {
        =:  (%, %) -> Boolean; ++ equality
        ~=: (%, %) -> Boolean; ++ inequality
}
```

This would imply that in order to satisfy this type a domain needs to export the two operations above. However, it seems a waste not to use the fact that inequality may be implemented in terms of equality in order to save every domain satisfying this category having to define both operations. This is achieved through *default implementations* (*default constant bindings*).

```
with {
        =:  (%, %) -> Boolean; ++ equality
        ~=: (%, %) -> Boolean; ++ inequality
        default {
                (a: %) ~= (b: %): Boolean == not (a = b);
        }
}
```

The default implementation will call the "=" operation from the domain, then return the complement.

A (rather trivial) domain satisfying this category is:

```
Dom: with {
        =:  (%, %) -> Boolean; ++ equality
        ~=: (%, %) -> Boolean; ++ inequality
        default {
                (a: %) ~= (b: %): Boolean == not (a = b);
        }
 } == add {
        (a: %) = (b: %): Boolean == true;
 }
```

This domain suffers from the deficiency that it is impossible to create new elements (due to the definition of equality), but importing from it will add the constants "`=`" and "`~=`" to the current scope.

A domain is also free to implement operations which have default implementations. In this case, the domain *over-rides* the operations in the default and importing the domain will activate the operations in the domain.

In a default definition, the uses of other exports from the type are obtained by looking up the operations in "`%`". This will first yield values from definitions in the domain or more closely applicable default bodies. Thus, default implementations provide a mechanism for late-binding of names to values.

Defining category-valued constants

It is inconvenient to have to repeat category expressions in a program. The language allows categories to be treated as normal values and allows names to refer to categories. A category (by definition) is a value of the Aldor built-in type `Category`.

To decide whether a particular domain satisfies a category, it is necessary to know that category's value. For this reason, it is most useful to use the "`define`" keyword in giving categories and category-returning functions their values. This makes the value as well as the type publicly visible.

For example,

```
define Finite: Category == PrimitiveType with {
        #: Integer;
             ++ Number of values in the type.
        ...
}
```

defines a category, `Finite`, which exports a constant called "`#`", plus some other things. The following definition makes use of the `Finite` category:

```
NotALot: Finite == add {
        #: Integer == 0;
        ...
}
```

which creates a new domain "`NotALot`" for which the `#` constant has value 0. The remainder of the definition is elided here.

The new domain can then be used:

```
import from NotALot;
stdout << #$NotALot << newline;
```

The above program will print 0.

The new domain can also be used in contexts requiring something that satisfies `Finite`:

```
-- define a function giving the size of a domain
sizeof(FiniteDom: Finite): Integer == #$FiniteDom;
-- call it
sizeof(NotALot)       --- '0'
```

A category can be used inside a "`with`" body. Including a category places all the declarations of that category into the new category. This mechanism allows categories to inherit from one another.

Thus, we can define a new category, "`FiniteGroup`" as:

```
    define FiniteGroup: Category == with {
            Finite;
            1:   %;             ++ Identity for multiplication.
            *:   (%, %) -> %;   ++ Multiplication.
            inv: % -> %;        ++ Inverses.
}
```

In order to satisfy this category, a domain must implement at least all the non-defaulted declarations from `Finite`, as well as the three explicitly mentioned by `FiniteGroup`.

A "`with`" expression is a valid right hand side of a category definition. This allows the creation of parameterised categories. For example, the decomposed matrix category example above could be written as:

```
define DecomposedMatrixCategory(R: ArithmeticType): Category == with {
        decompose: Matrix R  -> %;
        solve: (%, Vector R) -> Vector R
}
```

A domain which satisfies this category may be defined as follows:

```
 DecomposedRationalMatrix: DecomposedMatrixCategory(Integer) ==
 add {
         Rep == ...

         decompose(m: Matrix Integer): % == ...
         solve(dm: %, v: Vector Integer): Vector Integer == ...
}
```

A domain producing map can also satisfy this category. For example:

```
Decomposed(R: ArithmeticType): DecomposedMatrixCategory(R) ==
add {
        Rep == ...

        decompose(m: Matrix R): % == ...
        solve(dm: %, v: Vector R): Vector R == ...
}
```

Join

The "Join" function takes as argument a tuple of categories and creates a new category which has the union of all their exports. Any conditions on declarations are `or`ed together.

For example,

```
Join(OutputType, PrimitiveType);
```

produces the category:

```
with { OutputType; PrimitiveType }
```

which includes all the exports from both OutputType and PrimitiveType.

Has expressions

A "has" expression has the following form:

dom has *cat*

where *dom* is a domain-valued expression, and *cat* is a category-valued expression. A "has" expression may be used in any part of a program, but is most often used to conditionalise domains and categories. The result of the expression is a `Boolean` value which will be true if *dom* can be shown to satisfy the category, and false otherwise.

Some examples:

```
Integer has ArithmeticType                                      -- true
Integer has FloatType                                           -- false
Integer has with { +: % -> % }                                  -- true
Integer has with { +: Integer -> Integer }                      -- true

Integer has OrderedArithmeticType with { factorial: % -> % }    -- true
Integer has OrderedArithmeticType with { bozo: % -> % }         -- false

List Integer has ListType Integer                               -- true
List Integer has ListType DoubleFloat                           -- false
```

The evaluation of this expression is made at run-time, so one may conditionalise code on the parameters to a function:

```
move(V: Vehicle): () == {
        if V has PlaneType then
                takeOff V;
        else if V has BoatType then
                sailAway V;
        else
                roll V;

}
```

This function will call “`takeOff`”, “`sailAway`” or “`roll`”, depending on the type-valued parameter V.

```
move(MountainBike)      --- calls roll
move(Concorde)          --- calls takeOff
move(Yacht)             --- calls sailAway
```

Conditional expressions

Frequently, a domain will satisfy additional categories if particular conditions on parameters, or on the domain itself, are met. This information may be incorporated into a category expression as a *conditional* statement. A conditional statement has the same form as an `if` statement, except that if the body contains declarations and definitions, they are associated with the condition for the purposes of type satisfaction.

For example,

```
with {
        BoundedFiniteLinearStructureType S;
        if (T has TotallyOrderedType) then {
            TotallyOrderedType
            sort!: % -> %
        }
}
```

is a category which is satisfied by any list type which exports an ordering on itself whenever its elements do. Provided that

```
S == Integer;
```

the following domain would satisfy this condition:

```
List Integer add {
        Rep == List Integer;

        local tails(l: %): Generator % == generate {
            while l repeat { tmpl := rest l; yield l; l := tmpl; }
        }

        (l1: %) < (l2: %): Boolean == {
                local x, y: %;
                for free x in tails l1
                    for free y in tails l2 repeat {
                        x.first > y.first => return false;
                        x.first < y.first => return true;
                }
                if not empty? x then false;
        }
        sort(l: %): % == {
                ...
        }
}
```

Here, the condition is satisfied because `S` is `Integer`, a member of the category `TotallyOrderedType`, and the appropriate operations are defined by the add body.

Conditional statements are most often used in parameterised categories:

```
define ListCat(T: Type): Category == with {
        if T has PrimitiveType then PrimitiveType;
        BoundedFiniteLinearStructureType T;
        ...
}
```

Evaluation rules

The bodies of "`default`" statements inside `with` and `add` expressions may include side-effecting statements. These will be evaluated in order to make the constants inside the body well defined, but the language makes no guarantees on when (or indeed, if) these side-effecting statements will be evaluated for a given category or domain.

The bodies of `with` and `add` expressions are evaluated in such a way that they will be evaluated after the expressions they depend on. If mutually recursive type-forming expressions are found within the body of either `with` or `add` expressions, the expressions are computed as a fixed point, rather than evaluated in strict sequence. This fixed point computation uses a technique from functional programming to create self-referential data structures.

CHAPTER 8

Name spaces

For a computer, or a human being for that matter, to understand an Aldor program it is necessary to establish the context which gives the meanings of symbols. This is perhaps more important than in other familiar programming environments, since Aldor has fewer built-in assumptions. Much of what is built-in with other programming languages is provided by libraries in Aldor. For example, Aldor libraries define the types "`String`", "`Integer`", and "`DoubleFloat`" and their operations. Typically, the interface to these libraries is through an include file which imports the library so that it may be used in the current program.

This chapter describes how symbols in a program are associated with particular meanings, and how a meaning is selected when several are applicable.

8.1 Scopes

A symbol's meaning is given by the context in which it appears. A particular meaning (for example: "`n` is the second parameter in the definition of the '`+`' function in the domain `Integer`") has a visibility, or scope, governed by the constructs in which it is introduced. In common with most programming languages, Aldor mainly uses lexical scoping.

New scopes in Aldor are introduced by the following expressions:

- *E* `where` *Definitions*
- `+->`
- `with`
- `add`
- `for` *i* `in ...`
- Applications, *e.g.* `Record(i: Integer == 12)`

These forms may be nested to any depth. Note that the last two bind names in particular positions in the expression, and do not form general scope levels.

Lexical scoping implies that the only variables visible at a given point in a program are those that have been created locally or imported into scopes surrounding the current point.

In Aldor, there are two types of meaning for a given symbol — it must either be a constant or a variable. Constants are created in the following ways

- "`==`" statements.
- The bound symbols in a "`for`" iterator
- implicitly via an "`import`" statement
- implicitly via a declaration

A constant may not be assigned to, and therefore holds the same value throughout its lifetime. If two constants have identical names in the same scope (the name is *overloaded*), then an assigned variable's type or a qualification (using `$`) is used to disambiguate the uses of the constants.

A variable may be created explicitly, by a declaration, or implicitly, when it appears on the left of the assignment operator, "`:=`". Variables may be re-assigned; they may not be used in type-forming expressions.

When a scope forming expression is used in Aldor, all definitions and declarations directly within that scope are visible throughout the scope — sequences have no effect on what names are in scope. In the example

```
...
{
  x: Integer := 4;
  y: Integer := 3;
}
adds(a: Integer): Integer == x + a + z;
z: Integer := 3 + y
...
```

the current scope is extended with the variables `x`, `y` and `z` along with the constant `adds`. Note that `adds` uses `z` before it is defined in the outer sequence, and that `x` and `y` are defined in a subsequence, but the definitions are at the same scoping level as the others in the example.

A type may be viewed as an environment, mapping constant names and types into value bindings from a particular type. In Aldor, object files and libraries are values which map names and types into the values defined inside the file or library. This idea allows types, object files and libraries to be treated uniformly.

8.2 Constants

A constant definition may appear at almost any point in a program. If its outer defining scope is a `with` or `add` then it will be treated as an export of that type. If the type is used in a context not requiring the export, then the export will not be visible when the type is imported. A definition returns the type of the new variable.

A particular name in a scope may be overloaded with several constant values, either defined in the local scope or imported into it.

```
#include "aldor"
#include "aldorio"

x: Integer == 3;
x: String  == "hello";
stdout << (x*x)       << newline;     -- uses x: Integer
stdout << concat(x,x) << newline;     -- uses x: String
```

Here the name "`x`" refers to both a constant of type `String` and a constant of type `Integer`. In the two print statements, the constant to be used is selected according to context, in this case according to the available signatures for `+` and `concat`. In the first case there is no signature for `+` which takes a string as an argument, but there is one which takes two integers and returns an integer. In the second case there is no `concat` which takes an integer as an argument, but there is one which takes two strings and returns a string.

8.3 Disambiguators

Occasionally, it is not possible to tell which constant to use given a particular name. For example

```
x: Array Integer == [3,2,1];
x: List String  == ["Hello","my","friend"];
a: MachineInteger := 2;

stdout << x.a << newline;
```

In this case, "`x.a`" is ambiguous, as it may refer to either the `apply` operation from `List String` or the `apply` from `Array Integer`. In this case, we can specify which is required by *restricting* theresult to be of an appropriate type. If the second `String` of the list (note that `List`s indexing starts at 1 while `Array`s start at 0) is wanted then the print line should read

```
stdout << (x.a)@String << newline;
```

Sometimes a constant with the same name and type may be imported from different domains. In this case the package call operator, `$`, can be used to disambiguate the constants. For example, both `SingleFloat` and `DoubleFloat` export a constant "`max`". Thus

```
stdout << max$SingleFloat << newline;
stdout << max$DoubleFloat << newline;
```

might print 3.40282320e+38 and then 1.79769313486231467e+308 on a particular machine.

8.4 Import from

The "`import`" statement brings constants which are exported from types into scope[1]. The simplest form of an `import` statement is

`import from` $D_1, \ldots, D_n$`;`

with $n \geq 1$. This form imports all of the exported constants from the given domains (these are treated as type-forming expressions, see chapter 14). The importations are made in sequence, so that later domains can depend on exports from earlier ones. The precise exports of types are given by the category of the type, as described on page 92.

The complete form of the `import` statement is:

`import` *`restrictions`* `from` $D_1, \ldots, D_n$`;`

This form is used less often than the previous one, typically when a small number of operations are required from a group of types. Importing the "`<<`" operations from a type is a common example.

This form introduces the exported constants from $D_1, \ldots, D_n$ into the current scope. In this case, the restrictions are either a category expression or a sequence of declarations.

```
import { +: (%, %) -> %; -: % -> % }
      from Integer, String, Float;
import ArithmeticType from DoubleFloat, Integer;
```

The first statement will import the constants for addition and negation from `Integer` and `Float` into the current scope. Nothing will be imported from `String` as this type does not export either operation.

The second will import the operations necessary to satisfy `ArithmeticType` from the given domains.

Values created by programs written in other languages can be made visible using an import from a "`Foreign`" type. This is described in Section 14.15.

[1] Libraries in Aldor are simply types which export their contents.

8.5 Inline from

The "`inline`" keyword is similar to the "`import`" keyword, but instead of importing names into a scope, it allows the result of compiling the current scope to depend on the compiled values of the constants indicated. This dependency information may then be used by the compiler to determine if a particular function may be inlined (for optimisation) instead of called in the normal way. Allowing this form of optimisation can give very high quality code. The cost is that, if a file is recompiled, then every file which contains an `inline` statement mentioning that file should also be recompiled to ensure consistency, slowing down the development cycle.

```
#library AldorLib "aldor"

import from AldorLib;
import from Integer;
import from DoubleFloat;
import from TextWriter;          -- For ‘stdout and ...
import from Character;           -- for ‘newline’ and ‘<<’
                                 -- (all normally imported
                                 -- when ‘aldor’ included).
inline from Integer;

stdout << 3 + 2     << newline;  -- 3 + 2 gets optimised.
stdout << 3.0 + 2.0 << newline;  -- 3.0 + 2.0 doesn’t.
```

(See section Section 17.2 and section Section 17.3 for an explanation of "`#library`".) This code, when compiled with optimisation on, converts the function calls from the `Integer` domain into machine-level operations, but leaves the other operations as ordinary function calls.

One consequence of granting permission to inline from a domain is that permission is also granted to inline from all types exported by the domain. So using `inline from AldorLib`, for example, grants permission to inline from *all* the domains in the above example (and the rest of the Aldor library).

There are two other consequences of inlining: first of all it can increase the size of the compiled code, and secondly it can make debugging harder. For these reasons no inlining is done automatically, and there are a number of compiler flags which the user can use to either enable inlining of any function, disable all inlining, or restrict the increase in code size by a particular amount (see the documentation for `-Q inline`, `-Q inline-all` and `-Q inline-limit` in Section 23.8).

As in the `import` statement, there may also be restrictions on the particular constants allowed to be inlined.

8.6 Variables

New variables are created by declaration statements (described below), or implicitly by the first assignment to a variable inside a scope. In the implicit case, the variable is lexical and local to the scope.

It is an error to have two variables with the same name in the same scope—thus

```
x: Integer := 3;
x: String  := "hello";
```

will give a compile-time error "Variables cannot have different types in the same scope". In addition, it is an error to define a constant with the same name as a variable within a scope. An assignment will create a new variable hiding any names not explicitly imported. For example:

```
import from Integer;

-: Integer := 3;
```

defines a new variable of type `Integer`. The "`-`" function from the domain `Integer` is hidden by this statement — it can still be accessed by using the qualified name: `-$Integer`.

Aldor will generate a warning if an implicitly local variable in a new scope shadows a similarly named variable in an outer scope.

8.7 Functions

A function introduces a new scope level which includes the parameters to the function: As you might expect, parameters to the function are visible inside the function. A function expression has the following form:

```
(s1: S1 == v1, ..., sn: Sn == vn) : (t1: T1, ..., tm: Tm) +-> E
```

where the expression `E` is treated as being in a new scope, with `s1`, ..., `sn` being introduced into that scope. The value of such an expression is a function which, when called with arguments `a`*i*, of appropriate types, will return the result of evaluating E with the the actual argument values `a`*i* substituted for the formal parameters `s`*i*. See the rules described in Section 6.5 for how this expression is evaluated.

The resulting function is sometimes known as a *closure*, as it closes over (*i.e.* gathers up, and places somewhere safe) the lexical variables (not the values of the variables) that it references.

For example,

```
...
import from List Integer;
n := 2;
m := 3;
if cond() then
        f := (a: Integer): Integer +-> n+a;
else
        f := (a: Integer): Integer +-> m*a + n;
m := 22;
return map f lst;
```

When the function-valued variable `f` is passed into the function `map`, the value of `m` used is 22 — not the value 3 which was in effect when `f` was defined.

Parameters

A function parameter may be assigned to, in the right hand side of a "`+->`" expression, where it is an implicit local; the right hand side of the "`+->`" expression is a fresh scope.

Parameters may be updated as variables. However, if they are not modified within the scope of the function, then they may be used in type-forming expressions (*e.g.*, expressions used in import statements).

8.8 Where

A "`where`" expression is of the form:

Expr `where` *Defns*

in which *Defns* is a sequence of declarations and definitions, used in the evaluation of *Expr*. For example

```
x+y where { import from Integer; x := 2; y := 3}
```

evaluates to 5. This can be useful when an expression has many repeated parts which can be factored out as a sequence of definitions. The names introduced in the declarations are visible in the expression part and also in the declarations (note that this expression does not import bindings from `Integer` into the outer scope). However, names introduced in the *Expr* are treated as if they are declared at the outer scope level, so

```
x: Integer == y where { import from Integer; y := 2}
```

adds a variable "`x`" to the outer scope, the definition of which references "`y`" which will not be visible in the outer scope.

8.9 For iterators

A "`for`" iterator introduces a new local name, unless that name is declared free (see Section 5.14). The name is local to the "`repeat`" loop or collect form, and is treated as a constant. That is, it may not be updated within the body of the loop or collect expression.

8.10 Add

The "`add`" operator has the following syntax:

Add-domain `add` *declarations*

It combines a group of declarations with a (possibly omitted) domain, to form a new type (see Section 7.8). Declarations on the right hand side of the `add` are marked as being exports of the new type, provided that they are not explicitly defined as local.

An `add` expression also introduces a binding for the constant `%`, which is a reference to the domain formed by the `add` expression.

8.11 With

A "`with`" expression forms a new category, and has the following syntax:

L `with` *R*

This is equivalent to

`with` { *L* ; *R* }

where *L* evaluates to a category and *R* is a sequence of either declarations or other category expressions. These form a new scope, which also contains a binding for `%`, which refers to the domain over which the category is built (that is, the domain under consideration in the category) and whose type is the value of the `with` expression. (For more details on categories, see Section 7.9.)

For example,

```
#include "aldor"

define BinaryAggregate: Category ==
    Join(BoundedFiniteLinearStructureType(Boolean), BooleanArithmeticType) with {
                     default {
                               ~(barr: %): % ==
                                     [not bit for bit in barr];
                               (a: %) /\ (b: %): % ==
                                     [b1 and b2 for b1 in a for b2 in b];
                     }
     }

DumbBitArray: BinaryAggregate == Array Boolean add {
     (\/)(x: %, y: %): % == ~( (~(x) /\ ~(y)));

     xor(x: %, y: %): % == (x /\ ~y) \/ (~x /\ y);

     (=)(x: %, y: %): Boolean == {
                for bitx in x for bity in y repeat
                         if bitx ~= bity then return false;
                true;
     }
}
```

Defines a new category (the `define` keyword is explained in Section 8.13) called `BinaryAggregate`. The type `DumbBitArray` is a simple domain satisfying `BinaryAggregate`.

The `%` in the body of the `with` statement refers to a domain of type `BinaryAggregate`.

A `with` expression also defines a constant named "`%%`" for *each* category from which the `with` expression inherits. The type of `%%` is the inherited category, and the value is the domain viewed as a member of that category. In the example, `%%` bindings are in scope for the following categories: `BinaryAggregate`, `BooleanArithmeticType`, `BoundedFiniteLinearStructureType Boolean`, `BoundedFiniteDataStructureType Boolean`, `FiniteLinearStructureType Boolean`, `PrimitiveType`, and so on. The `%%` bindings are generally most useful for checking conditions.

8.12 Application

Applications allow arguments which are declarations or definitions. The identifiers which are declared or defined in arguments are then local to the application form. For example, in

```
F(3.2, 5.8, tolerance == 0.02)
G(T: Type, List T),
```

the identifier `tolerance` is local to the application of `F`, and the identifier `T` is local to the application of "G".

A comma expression may declare identifiers to be used in later elements. In the expression $(e_1, \ldots, e_n)$, if any e_i is a declaration or definition, then the name is visible in the expressions e_j, $i < j \leq n$. The declarations place names in the current scope — the comma expression of itself does not create a new scope. So, in the case where a comma expression provides arguments to a function, declared identifiers are local to the application.

This allows dependent function types to be created: the `->` operator is simply a function which is applied to two tuples of types. For the expression $(S_1, \ldots, S_n)$ `->` $(T_1, \ldots, T_m)$ any identifiers declared in S_i are visible in any of the T_j. In addition, both the left and right hand sides are comma expressions, so the above rules apply.

The following is a possible example of the use of dependent types to form a new function type:

```
f: (T: Type, t: T) -> (LT: ListType T, lt: LT)
```

Note that "`T`" is used on both sides of the arrow.

It is similarly possible to create dependent product types: the `Cross` operator is a function which accepts some number of types as arguments and produces the product type. The argument types may have declarations which induce dependency. For instance:

```
Cross(T: Type, List T)
```

While the built-in functions "`->`" and "`Cross`" allow dependency inducing declarations as their arguments, the language does not currently provide a mechanism for creating new functions which support this[2]. Thus, although

[2] Making this the default for *all* functions would add a significant run-time cost to almost all function applications in a program.

```
HashTable(n: Integer, IntegerMod n)
```

is a legal call, `HashTable` as a type defined in a library cannot support this form of dependency, and an error may be signalled by the compiler when this type is used.

8.13 Declarations

Declarations associate a type with a name. A declaration is of the form:

modifier idlist: *Type*.

The *modifier* is one of:

- default
- define
- local
- fluid
- free
- export

Either (but not both) of the modifier and the "`:` *Type*" may be omitted. A declaration may appear in any context not requiring a value, and remains in force throughout the enclosing scope. If the modifier is omitted, the compiler assumes that `local` is meant and issues a warning that `default` may be intended.

Some modifiers allow definitions or assignments in a declaration. In this case, the type part is optional and the declaration has one of the forms

Modifier id [: Type] := E
Modifier id [: Type] == E

If the type information is omitted, then the type is inferred or taken from `default` declarations. When the type information is present, say declaring *id: T*, the declaration also imports type *T* into the current scope. If this is not desired, then the declaration may use "`:*`" in place of "`:`" to avoid the import.

Finally, it should be noted that it is also possible to give a sequence of assignments or definitions in these statements. For example,

```
local {
        a: Integer := 1;
        b: Integer := 2;
        c: Integer := 3
}
```

Default The "`default`" modifier declares that any instances of the names specified will have the type indicated. This does not create any new bindings.

```
default n: Integer;
n := 23;
f(n): Integer == n + 1;
stdout << n^10 << newline where { local n := 2 };
stdout << n    << newline;
```

This example creates 3 integer variables named "`n`". The type of these does not need to be specified as it is given in the default statement. This example prints 1024, and then 23.

The "`local`" declaration in the "`where`" statement is included to avoid a warning about the n in the outer scope.

A warning is given if a binding has a default type and there is a declaration in scope with a second type.

A default statement around a definition or definition sequence inside a "`with`" scope modifies the way that exports from the current domain are interpreted. This allows generic methods to be defined which employ definitions found in inheriting types. This is further explained in Section 7.9.

Define A "`define`" modifier allows the definition of a value to be visible as well as its type. This is especially useful in category-forming definitions, because without the `define` it is impossible to decide what signatures are exported by the category.

```
define Monoid: Category == PrimitiveType with {
    1:      %;           ++ Identity for multiplication.
    *:      (%, %) -> %;  ++ Multiplication.
}
```

The above example defines the category `Monoid`. Without the "`define`" keyword, uses of this definition would only have the type of the category (`Category` in this case) available.

Local The "`local`" modifier declares that the given identifiers are local to the current lexical scope. For example `local x` declares that the "`x`" will be a local, and does not specify a type, so this will be deduced at the first assignment to the variable. A `local` declaration may also include an initial assignment or definition of the names it introduces.

Names which are assigned using "`:=`" and not otherwise declared are treated as `local`.

Fluid The "`fluid`" declaration declares that the given identifiers should be treated as having dynamic, as opposed to lexical scope. The declaration is enforced within the lexical scope containing the declaration. Refer to section Section 8.14 for more details.

Free

A "`free`" declaration indicates to the compiler that the given name references a variable in an outer scope, and that the initial assignment to the variable should be interpreted as an assignment to the outer variable, rather than an initialisation of a new variable.

```
callCount := 0;
f(n: Integer): () == {
     free callCount;
     callCount := callCount + 1;
     n + 1;
...
```

The code above counts the number of times the function "`f`" is called. Without the `free` declaration for `callCount`, `callCount` inside the function would refer to a new local variable shadowing the outer variable. The free declaration may refer to either a parameter or a (possibly fluid) variable.

Export

An "`export`" modifier may be used to declare that certain names are to be made visible outside the scope in which they are defined. This is the effect when `export` is used at the top level of an "`add`", "`with`" or file scope. In other contexts, `export` has the same meaning as `local`.

An `export` declaration may be followed by an optional "`to`" part. This is used to make Aldor values visible to programs written in other programming languages. See Section 14.15 for details.

An `export` declaration occurring in a `with`-expression may be followed by an optional "`from`" part. This indicates the source of the items to be exported, in the same way the "`from`" part of an `import` or `inline` statement indicates the source of the items to be imported or inlined. This is described in Section 7.9.

Names which are defined using "`==`" and not otherwise declared are treated as `exports`.

8.14 Fluid variables

Fluid variables are not often needed but can be useful when a large amount of dynamic state is needed, or a routine is parameterised by a very large number of variables.

A fluid variable exists throughout the lifetime of a program, and its value is always the most recent extant binding of the variable. The extant bindings are the bindings of the variable inside `fluid` declarations from scopes which have not yet been exited.

When a variable is declared fluid, all references to that name *inside* the declaration's scope are assumed to be fluid.

An example might help:

```
#include "aldor"
#include "aldorio"

fluid n: Integer := 2;

f(): () == stdout << "The value of n is " << n << newline;
g(): () == { fluid n := 3; f() }

f();
g();
```

The fluid variable "n" is bound at the top-level and given the value 2. This is the value printed by the top-level call to f. In the next call, the function g re-binds n giving it the value 3. Then when f is called, it is the new value, 3, that is printed. On exit from g, this binding of n is removed and n assumes the value it had in the outer scope.

As usual, an inner declaration (*e.g.* local, export) may locally override an outer declaration (*e.g.* fluid). Note that without the "fluid" declaration in g, the inner n would be treated as having an implicit "local" declaration. This would then behave as in the example below. The inner occurrence of n is a new local variable, unrelated to the outer n, and the call to g results in 2 being printed.

```
#include "aldor"
#include "aldorio"

fluid n: Integer := 2;

f(): () == stdout << "The value of n is " << n << newline;
g(): () == { local n := 3; f() }

f();
g();
```

If no initialisation is given in the fluid declaration, then the variable is taken to exist in an outer dynamic scope. In the example below, the function "g" provides a binding for the variable "n". Then, when the function f is called from g, the uses of "n" in f refer to the binding in g.

```
#include "aldor"
#include "aldorio"

f(): () == {
        fluid n: Integer;
        stdout << "The value of n is " << n << newline;
}
g(): () == {
        fluid n: Integer := 3;
        f();
}

g()
```

If an assignment to an existing fluid variable occurs in a context other than a `fluid` declaration, it will modify the current value of the variable, rather than creating a new one:

```
#include "aldor"
#include "aldorio"

fluid n: Integer := 2;

f(): () == stdout << "The value of n is " << n << newline;

g(): () == {
        fluid n: Integer;
        n := n + 1;
        f()
}

g();
g();
```

A file may use fluid variables which have been bound in other files, but no type information regarding these variable is known, so it is the programmer's responsibility to ensure that the types match[3].

In the current implementation of Aldor, a fluid variable must have a binding point at the top level of some file. It is an error to have two fluid variables of the same name and different types in a program.

[3]Compare Section 7.1.

CHAPTER 9

Generators

Consider the problem of traversing a list to operate on each of its members. One might write code such as:

```
-- Approach 1: tailing
t := L;
while not empty? t repeat {
        a := first t;
        t := rest t;
        f a
}
```

This approach makes good sense for linked lists, but is too expensive for lists represented as arrays. Each iteration would have to allocate a new array in the call to `rest`, leading to $O(\#L^2)$ storage use where none is really necessary.

Alternatively, one could write:

```
-- Approach 2: indexing
for i in 0..(#L - 1) repeat {
        a := L.i;
        f a;
}
```

This approach makes good sense for lists represented as arrays, but is inappropriate for a linked list representation. Each iteration would need to traverse the list from the beginning to find the desired element, leading to $O(\#L^2)$ time where $O(\#L)$ should suffice.

Having seen this, consider the problem of writing a generic program to operate on some `BoundedFiniteDataStructureType` structure which might be represented as an array, as a linked list, or in some other way.

How should loops be written to minimise cost in a generic program?

A related problem arises with more sophisticated data structures. Here, the steps to traverse an object can be rather intricate. The code for a loop which traverses objects in parallel can be extremely difficult and error-prone.

How can loops be written which hide the complexity of data representation?

The answer to both these questions is the same in Aldor, and that is to use generators.

9.1 Using generators in loops

Generators may be used in Aldor to obtain values serially, wherever required. For example: to compute numbers in a sequence; to access the components of a composite structure, such as a list, array or hash table; or to read data from a file.

The simplest way to use a generator in Aldor is with a "`for`" iterator on a repeat loop or a collect form:

```
#include "aldor"
#include "aldorio"

import from MachineInteger;

-- Generators in a for-repeat loop.
import from Generator MachineInteger;
g := generator(1..10);

for i in g repeat { stdout << i << newline }

-- Generators in a for-collect loop.
import from List MachineInteger;
h := generator(1..10);

l := [ i*i for i in h ];

stdout << l << newline
```

In fact, this form of using generators is so common, that if the expression E in "`for` v `in` E." does not belong to a generator type, then an implicit call is made to an appropriate "`generator`" function. This is equivalent to writing "`for` v `in generator` E."

With this, our example may be written as:

```
#include "aldor"
#include "aldorio"

-- Implicit generators in a  for-repeat loop.
import from MachineInteger;

for i in 1..10 repeat { stdout << i << newline }

-- Implicit generators in a  for-collect loop.
import from List MachineInteger;
```

```
l := [ i*i for i in 1..10 ];

stdout << l << newline
```

In Aldor *all* `for-repeat` and `for-collect` loops are based on generators. There is no built in method to traverse integer ranges, lists or other structures. It is the compiler's responsibility to make the use of generators reasonably efficient.

Generators are normal values in Aldor and, once created, may be passed as arguments to functions which consume them gradually, according to a cooperative scheme.

9.2 Using generators via functions

Sometimes it is desired to use the values in a generator gradually, with some logic that is not naturally expressed as a `for` loop.

One might imagine writing a function, such as the following, to extract elements from a generator one at a time.

```
first(g: Generator S): Union(value: S, nil: Pointer) == {
        for s in g repeat return s;
        nil
}
```

Here the loop body would be evaluated zero or one times, and the function would return `nil` or the first value in the generator. Subsequent calls would extract the remaining values.

The standard Aldor library provides a type with a single function called `next!`. Each time this function is called, it returns the next available value in the generator. When no more value is available, a `GeneratorException` is thrown. To use this function one must have an include command for `libaldor` and import from the appropriate generator type, *e.g.*:

```
#include "aldor"
import from Generator T;
...
```

With this, the following function becomes available:

- `next!`: `Generator T -> T`

The function "`step!`" runs the generator code until the next value, or the end of the generator, is reached. After `step!` has been called, the function "`empty?`" may be used to test whether the generator has been exhausted. If `empty?` returns `false`, then the "`value`" function may be called to extract the current value from the generator.

The expression "`for x in g repeat E`" is equivalent to

```
try
   repeat {
        x := next! g;
        E
   } where { local x };
catch EXCEPTION in {
   EXCEPTION has GeneratorExceptionType => {}
}
```

9.3 Creating generators

Generators are ultimately created with a "`generate`" expression in one of the following forms:

`generate` E
`generate to` n `of` E

The first form, without the "`to`" part, is the most commonly used.

The body, E, of a `generate` is an expression containing some number of "`yield`" forms, each with some argument. The evaluation of the `generate` proceeds as follows:

None of the expressions in the body is evaluated when the generator is first formed. When the first value is requested from the generator, the body is evaluated until a `yield` is encountered. The argument of the `yield` is returned as the value of the generator, and evaluation of the generator is suspended. When another value is requested from the generator, evaluation resumes from the point where left off, and continues until the next `yield` is encountered. Evaluation proceeds in this way, suspending at yield points and resuming when further values are requested, until the evaluation of the body expression is complete. Note that some evaluation may occur after the last yield, before the body has finished evaluating.

All the `yield`s for a given `generate` must have the same type of argument. If all the `yield`s in a particular `generate` have type T, then the `generate` expression has type "`Generator(`T`)`".

If given, the "`to`" part provides a bound on the number of values which the generator may provide. If a bound is not given, the compiler is permitted, but not required, to deduce a bound when it can.

Examples:

`generate` expressions may have several `yield`s:

```
generate { yield 1; yield 2; yield 3 }
```

A `yield` may appear in a loop:

```
generate {
        while not empty? l repeat {
```

```
                yield first l;
                l := rest l;
        }
}
```

The generator body may have arbitrary control flow within it. This encapsulates the logic for traversing a structure. This is an example from the innards of the `HashTable` type in the base Aldor library:

```
generate {
        for b in buckv t repeat
                for e in b repeat {
                        c: Cross(Key, Value) := (e.key, e.value);
                        yield c
                }
        }
}
```

In the Base Aldor library, the generator for general `IntegerSegment` values (a..b by c) is given as:

```
generator(s: %): Generator S == generate to size s of {
        a := low  s;
        b := high s;
        d := step s;
        open? s =>              repeat (yield a; a := a + d);
        d >= 0  => while a <= b repeat (yield a; a := a + d);
        d <  0  => while a >= b repeat (yield a; a := a + d);
}
```

Since it might not be obvious, we note that recursive functions may be used to implement generators. This is extracted from the "`Tree`" example in Section 21.8:

```
preorder(t: %): Generator S == generate {
        if not empty? t then {
                yield node t;
                for n in preorder left  t repeat yield n;
                for n in preorder right t repeat yield n;
        }
}
```

Finally, we observe that when using the base library it is possible to form generators by providing a set of `empty?`, `step!`, `value` and `bound` functions. These would normally operate with a shared environment:

```
#include "aldor"
#include "aldorio"

GI ==> Generator Integer;
import from GI;

everyOther(g: GI): GI == {
```

```
        s!(): () == {step! g; step! g}
        e?(): Boolean == empty? g;
        v():  Integer == value g;
        b():  MachineInteger == {n := bound g; n = -1 => n; n quo 2}

        generator(e?, s!, v, b)
}

gg := everyOther generator(1..20);
for x in gg repeat stdout << x << newline
```

CHAPTER 10

Post facto extensions

One of the real problems in reusing code is how to use programs from independently developed libraries at the same time.

Every library has assumptions about the basic behaviour of the values it uses. To use values created by functions of one library as input to functions of a second library, it is usually necessary to convert the values to types derived from the second library.

As the number of libraries used by an application increases, the conversion problem becomes more significant. For example, say a library defines a concept of "`FancyOutput`", of which the basic `Integer` type could be an instance, if only it supported a few extra operations. The way this would be handled in other languages would be to derive a new type from `Integer`, say "`FancyOutputInteger`", and to use fancy integers in the program.

Now suppose we wish to use a library for differential operators, which defines a concept of "`DifferentialRing`". Now the integers would be a suitable instance, if only they provided a differentiation operator. It is a trivial matter to supply a differentiation operator for the set of integers: it would just return 0. At this point we end up with the new type "`DifferentiableFancyOutputInteger`" which may now be used in place of "`Integer`" when using the differential operator library at the same time as fancy output.

Aside from being ugly, this leads to a second problem: Each library may refer to the original type Integer, or its own derived version of it, in particular situations, and the `DifferentiableFancyInteger`s will need to be converted to `FancyInteger`s or `Integer`s and *vice versa*.

This problem gets worse when both libraries use the same base type, `String` or `Window` for instance, with different sets of operations.

This problem is particularly acute in certain applications. In mathematics, for example, certain basic sets are simple instances of very many abstract concepts. It is impossible for the Aldor library to anticipate all the mathematical contexts in which the type `Integer` may be expected to perform.

The way Aldor solves the problem just described is to allow programs to supply enhancements to already existing types via *post facto extensions*.

10.1 Extending types

If T is a type-valued constant, then its meaning may be extended with a definition of the form:

```
extend T: C == D
```

Here C is a new category to which T will now belong and D is a package providing implementations for the newly required exports. The new operations cannot see the internal representation of T.

Taking the earlier example, the type `Integer` can be made to satisfy the new categories `FancyOutput` and `DifferentialRing` by providing appropriate extensions:

```
extend Integer: FancyOutput == add {
        box(n: Integer): BoundingBox == [1, ndigits n, 0, 0]
}

extend Integer: DifferentialRing == add {
        differentiate(n: Integer): Integer == 0
}
```

then these extensions may be used, independently or together,

Extensions are made visible in the same manner as other definitions: by importing the package which provides them. This allows programs using extensions to be statically checked and more heavily optimised.

Extensions may be private to a particular application, or be exported by a library. In the case where a library provides an extension, the clients of the library see the extension applied to the type.

Within a given context, a type-valued constant has as its type the `Join` of the categories of the original value all the extensions. Its value is obtained by `add`-ing the extensions to the original value.

If two extensions have intersecting categories, then they may not be imported in the same scope.

10.2 Extending functions

In typical Aldor applications many of the types are provided by functions, so an effective extension mechanism must provide some way to extend these types as well.

One might imagine a solution which took a particular function result, say "`List(Integer)`" and modified it. The problem with this is that while that one list type would be extended, the other list types would not be. It is really the meaning of `List` which should be extended.

To extend the meaning of a function-valued constant in Aldor one writes:

`extend` *fdef*

where "*fdef*" has the form of a definition of the function being extended. For example:

```
#include "aldor"
#include "aldorio"

-- Partial(S) is a type which includes values in 'S' as well
-- as a special 'failed' value.
--
-- This extension does two things:
-- 1. It allows 'failed' to be treated as 'false' in 'if' statements.
-- 2. It causes an import from Partial(S) to also import from S.

extend Partial(S: Type): with {
        test: % -> Boolean;

        export from S;
}
== add {
        test(x: %): Boolean == not failed? x
}

PI ==> Partial Integer;
import from PI;

i := failed; j := [7];

stdout << (if i then "oops" else "ok") << newline;
stdout << (if j then "ok" else "oops") << newline;
```

Any form of function definition is allowed, *e.g.*:

```
extend F(S: Type): C(S) == ...;

extend G: (S: Type) -> C(S)  ==  (S: Type): C(S) +-> ...;

extend H(n: Integer, S: PrimitiveType) (R: ArithmeticType) == ....;
```

Function extensions are meaningful when the argument and result types of the extension are compatible with the argument and result types of the original meaning.

In the current implementation of the language, the extension argument types are compatible if they are the same as the argument types of the orginal function.

Corresponding result types are compatible if they are the same or both are subtypes of the same type. Additionally, the (base) type must have an appropriate method to combine values. At present only functions and types provide a combination method.

In practice this means that types, or functions producing types, or functions producing functions producing types, *etc.* may be extended.

Type values are combined by `add` and their types are combined with `Join`. Function values are combined by producing a new function which runs all of them and combines the result (recursively, if necessary).

10.3 Extending the base Aldor library

The base Aldor library builds many of its types in layers, using "`extend`". For instance, it extends `Boolean`, `Generator(S)` and `Tuple(S)`, which are language-defined, and creates `Integer` and `TextWriter` in stages.

An application which introduces a new concept can provide a file to extend an existing library to support the new concept.

Suppose, for instance, that an application needed to determine how much dynamically allocated memory was used by data structures. Then it could extend the Aldor library with a file which began with code resembling that in Figure 10.3.

With this, it would now be possible to ask about the sizes of values belonging to the types "`Integer`", "`List(Integer)`", and so on.

Figure 10.1: Post facto extension of the Base Aldor library.

CHAPTER 11

Exceptions

11.1 Introduction

What happens when something goes wrong during a call to a function? For example, the system detects an error in its arguments (*e.g.* divide by zero), or that it can't perform a particular operation (*e.g.* opening a file).

Using normal call and return code, you have two options: the function can cause the program to halt, *e.g.* by a call to "error"; or it can return a partial or union type which includes a flag to say that the operation failed. The former may be a bit 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 and check for the failure case, which may or may not be useful. It certainly gets long-winded if the failure case has to be propagated through a deep call-chain.

A more concise way of signalling error conditions is provided by the exceptions mechanism. 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 thrown by an expression
4. Defining exception objects

11.2 Throwing Exceptions

The syntax for raising an error is:

```
throw exception
```

where *exception* 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 throw expression is `Exit` (but see later) — *i.e.* no value is created by the expression, but control does not pass to the expression (compare this with the return statement, for example). A throw statement is therefore a way of causing a function to terminate abnormally.

11.3 Catching Exceptions

To catch an exception the syntax is:

```
try Expr catch id in handler finally 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 ZeroDivide(Integer) => ...
     E has BadReduction => ...
     E has FileException => ...
     true => throw E;
     never;
}
```

Where each item on the right hand side of the `has` denotes an exception type. The last line is to ensure that the statement compiles successfully.

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 ZeroDivide => 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, the *stmts* in the `finally` part of the try expression are evaluated. These are guaranteed to be evaluated even if the handler itself raises an exception.

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

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

Note that the '`catch` *id* `in ...`' part is also optional provided that a `finally` part is supplied.

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

11.4 Specifying Exceptions

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 types. Typically, the `throw` keyword is applied to the return types of functions to indicate which exceptions they can raise, for example:

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

This indicates that *foo* may only raise exceptions of types *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(): Integer == throw ZeroDivide;

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

Here `badIdea` indicates that it will not raise an exception, while `justDie` will always raise an error. The compiler may warn the user in this situation.

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

```
foo(): Integer 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
    }
}
```

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: Integer -> Integer 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: We simplified things earlier when we said that the type of an `throw` statement was `Exit` — the actual type of "`throw X`" is `Exit throw` *`typeof(X)`*. Where *typeof(*`X`*)* indicates the exact type of the exception expression. This indicates that flow of control stops, but the exception `X` may be raised.

11.5 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 `ZeroDivideException` may inherit from `ArithmeticException` for example), and the domain definition provides a mechanism for creating the exception.

For example,

```
define ZeroDivideException: Category == ArithmeticException with;
ZeroDivide: ZeroDivideException == add;
```

If `ZeroDivide` is defined this way, then any handler with a clause "`E has ArithmeticException`" will also catch `ZeroDivide` exceptions. There is an `is` keyword in the language if you need to do exact matching, but it shouldn't be needed that often.
This mechanism also allows one to create parameterised exceptions:

```
define ZeroDivideException(R:Ring):Category == ArithmeticException with;
ZeroDivide(R:Ring):ZeroDivide(R)@Category == add;
```

and to have values defined within the exception:

```
define FileException: Category == ArithmeticException with {
        name(): String;
}

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

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

```
foo(): () == {
        ...
        throw (add {name(): String == <bizzare-calculation>})_
@FileException
        ...
}
```

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 defines rather a lot of objects, is:

```
define FileException: Category == Exception with {
        name(): String;
}

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

FileError(vvv: String): FileException(vvv) == add;
```

CHAPTER 12

Generic tie-ins

Several syntactic constructs are defined in terms of function application in Aldor: the treatment of literal values, the application of one object to another, the updating of objects, the interpretation of tests in conditional statements and the mechanism for generating a set of values for iteration.

This allows user programs to define how these syntactic constructs behave in a particular lexical context. The function calls themselves are treated as standard function calls, so the normal rules apply.

12.1 Literals

Types provide meanings for literal constants by defining functions named "`integer`", "`float`" or "`string`" taking values of type `Literal`. The type `Literal` represents the source text of the particular literal. Valid literal values are described in Section 5.2.

The meaning of string-style literals is determined by what operations

```
string:  Literal -> X
```

are visible, where "`X`" may be any type. In `libaldor` the type `String` provides string-style literals.

Both integer and floating point literals are passed in the same way, so it is legal to call the function `string` on them. This can be used to allow numbers to be parsed as strings. For example, the function `scan`, from `IntegerTypeTools` in the Aldor base library, scans a `TextReader` stream to find a possible `Integer` value. The following example is a possible implementation to create an `Integer` from a `Literal`.

```
integer(l: Literal): Integer == {
        import from IntegerTypeTools Integer, String;
        scan (string l)::TextReader
}
```

Note that the literal "`1`" is converted to a string, and then to a `TextReader` input stream before `scan` is called on the result to form a new value.

12.2 Program-defined tests

There are several types of expression in which a condition controls the evaluation of an Aldor program:

```
if condition then ...
condition => ...
while condition repeat ...
for ...  in ...  | condition repeat ...
not condition
condition1 and condition2
condition1 or condition2
```

In many situations, a value can be treated as a condition, even though it may not be a value from the "`Boolean`" type. Aldor allows these to be treated as logical values in the above constructs. If a *condition* above produces a value of type T, different from `Boolean`, and there is a single function "`test:` T `-> Boolean`" in scope, then that "`test`" function is implicitly applied to the condition value to determine the outcome of the test. For example:

```
#include "aldor"

-- In "aldor" the type List(S) has a function "test" which
-- returns true on non-empty lists.

-- This function determines which of two lists is longer.
LI ==> List Integer;

longer(a: LI, b: LI): LI == {
        (a0, b0) := (a, b);
        while a and b repeat (a, b) := (rest a, rest b);
        if a then a0 else b0
}
```

12.3 Generator

If an expression traversed by a "`for`" iterator does not evaluate to a `Generator` value, then the operator "`generator`" is implicitly applied to the expression. This makes loops more readable if the `for` is traversing, for example, a list or an array:

```
#include "aldor"
#include "aldorio"

import from List Integer, Integer;

ls := [1,2,3,4];

for elem in ls repeat stdout << elem << newline;
```

The "`for elem in ls repeat ...`" of the previous example is equivalent to:

```
for elem in generator ls repeat ...
```

When `List Integer` is imported, then the application:

```
generator: List(Integer) -> Generator(Integer)
```

comes into the current scope.

12.4 Apply

In the absence of an explicit function named “`a`”, the application “`a(b)`” is treated as a call to the function “`apply`” with the first argument being taken to be “`a`”, and the remaining arguments being taken from the arguments to the original application. Example:

`f(a,b,c)` becomes `apply(f,a,b,c)`

For example, consider a matrix domain over a ring, R. A desirable syntax for retrieving elements of a particular matrix might be:

```
mat(a, b)
```

where `a` and `b` are integer indices for the matrix. To achieve this, a matrix domain would export a function with signature:

```
apply: (%, Integer, Integer) -> R;
```

The function is defined in the normal way.

12.5 Set!

If the left hand side of an assignment is an application, the assignment is treated as an application of the operator `set!` to the operator of the left hand side, the operands of the left hand side and the right hand side.

`f(a,b)`	`:= E`	becomes	`set!(f,a,b,E)`
`f a`	`:= E`	becomes	`set!(f,a,E)`
`f.a`	`:= E`	becomes	`set!(f,a,E)`
`f.a.b`	`:= E`	becomes	`set!(f.a,b,E)`

As an example, consider the matrix domain above. We would like to assign into the matrix using a syntax like:

```
mat(a, b) := 1;
```

To achieve this, the domain should export a function with signature:

```
set!: (%, Integer, Integer, R) -> R
++ update and return the previous value of the element
```

As this is just a normal function, there are no restrictions on the return type or value. In this case a value from R is returned, and the description indicates which. Note that the description is purely for documentation purposes, and not used to interpret the program.

12.6 Bracket

An expression of the form:

`[` *Expr* `]`

is treated as an application of the operator "`bracket`" to *Expr*.

Example:

```
#include "aldor"
#include "aldorio"

import from List Integer, Integer;

a := [1,2,3];
b := bracket(1,2,3);

stdout << a << newline; -- Produces: [1,2,3]
stdout << b << newline; -- Produces: [1,2,3]
```

As can be seen above, this is a useful syntax for creating aggregates of various types. The value passed to the "`bracket`" function in this case is a tuple of the three values 1, 2 and 3.

12.7 Coerce

An expression of the form:

Expr `::` *T*

is treated as an application of the operator `coerce` to *Expr* and restricted to be of type T:

`coerce(` *Expr* `)@`*T*

This allows types to define their own mechanisms for converting between types, and enables types to do appropriate error checking.

CHAPTER 13

Source macros

13.1 Macro definition

Aldor provides a way to define abbreviations, or *macros*, to make programs easier to read.

A common use of macros is to abbreviate names or type expressions, as in the example below. The lines containing "`==>`" are macro definitions:

```
-- Abbreviations for frequently used types:
MI  ==> MachineInteger;
L T ==> List T;

split(lmi: L MI): Record(first: MI, second: MI, rest: L MI) ==
        [first lmi, first rest lmi, rest rest lmi];

-- Abbreviations for long package names:
MStudy ==> LongitudinalStudiesOfMitralValveReplacementPatients;

d0 := intakes()$MStudy;
d1 := examinations()$MStudy;
```

The left hand side of a macro definition may be either an identifier or an application. A definition of the form

Op Parms `==>` *Body*

is equivalent to

Op `==> macro` *Parms* `+->` *Body*

and defines a parameterised macro. Any application syntax is permitted on the original left hand side: prefix, infix, or bracketed. The resulting right hand side is called a *macro function*.

This rule is applied until the left hand side is an identifier. Macros may consequently be defined to receive their arguments in groups, in a "curried" fashion. For example, when P_1 and P_2 are parameter sequences, the definition

Op `(`P_1`)(`P_2`) ==>` *Body*

is equivalent to

Op `==> macro (`P_1`) +-> macro (`P_2`) +->` *Body*

Macro functions may appear directly in the source program, and may be written with their parameters in groups:

`macro (`P_1`)(`P_2`)...(`P_n`) +->` *Body*

is equivalent to

`macro (`P_1`) +-> macro (`P_2`) +-> ...  macro (`P_n`) +->` *Body*

13.2 Macro expansion

The process or replacing the abbreviations with what they stand for is called *macro expansion.* Macro expansion works by making substitutions in the parsed form of programs. Because the substitution is on trees, it is not necessary to include extra parentheses in macro definitions.

Macro expansion transforms the source syntax trees with the following steps:

- Record and remove any new macro definitions encountered. A macro definition is scoped and persists for the remainder of the "`+->`", "`where`", "`add`" or "`with`" in which it occurs.
- Replace identifiers, if they have macro definitions, with the right-hand side of their macro definitions.
- Reduce expressions of the form
  `(macro` *Parms* `+->` *Body*`) (`*Exprs*`)`
  by introducing new macro definitions for the formal parameters and replacing the application with the macro expanded body.

Once macro expansion is complete, the entire program should be free of macro definitions and macro functions. The process of macro expansion removes all macro definitions; any remaining unreduced macro functions are reported as errors.

13.3 Examples

Given the following definitions,

```
a ==> A1 - A2;
b ==> B;
c ==> C;
d(e,f)(g,h) ==> (e+f)*(g+h);
x + y    ==> c(x,y);
f(g,x,y) ==> g(x,y) / g(y,x);
```

the following expressions expand as shown:

```
a;                                          --> A1 - A2
b;                                          --> B
a + b;                                      --> C(A1 - A2, B)
d(1,2)(3,4);                                --> C(1,2) * C(3,4)
f(+,u,v);                                   --> C(u,v) / C(v,u)
f((macro (a,b) +-> a), u, v);               --> u / v
(macro (a,b)(c)(d) +-> [a,b,c,d])(3,4)(5)(6); --> [3, 4, 5, 6]
```

13.4 Points of style

It is often convenient to use macros which are local to a function or expression. Below, we show a macro "l1?" which is local to the function "f".

```
f(li: List Integer, lp: List Pointer): () == {

        l1? x0 ==> (not empty? x and empty? rest x) where x := x0;

        if l1? li then ...
        if l1? lp then ...
        ...
}
```

The purpose of "l1?" is to provide an inexpensive test whether a list has length one. This example illustrates two additional points:

- Since macros are strictly syntactic substitutions the "l1?" macro may be applied to different values of different types.
- Since the macro uses the parameter in two places, the best practice is to introduce a new temporary variable so the argument is evaluated only once.

If the body of the macro is large, having a "where" at the end of a long expression can make a program harder to read. A convenient way to introduce local variables in this case is to make the left-hand side of the "where" a local macro.

```
BigBurger(patty1, patty2) ==> BB where {
        p1 := patty1;
        p2 := patty2;

        BB ==> {
                if not allBeef? p1 or not allBeef? p2 then
                        error "Bad food";

                b := burger();
                b << bottomBun;
                b << p1;
                b << cheese;
                b << lettuce;
                b << onions;
                b << sauce;
                b << middleBun;
                b << p2;
                b << cheese;
                b << lettuce;
                b << onions;
```

```
                    b << sauce;
                    b << topBun;
                    b
          }
}
```

Just as a macro function

`macro` *Parms* `+->` *Body*

is analogous to a function "*Parms* `+->` *Body*", a macro definition

Lhs `==>` *Expr*.

is analogous to a definition "*Lhs* `==` *Expr*". For this reason, a macro definition may be written in the equivalent form

`macro` *Lhs* `==` *Expr*.

CHAPTER 14

Language-defined types

The Aldor language defines only those types which are required in specifying what the language does. Most of the types which are usually found in high-level programming languages are delegated to libraries in Aldor. This allows the library designer maximum flexibility in dressing the basic types with desired operations.

The language defined types are listed below. Here, $n, m \geq 0$.

- Type
- (S_1,...,S_n) `->` (T_1,...,T_m)
- Tuple T
- Cross(T_1,...,T_n)
- Enumeration(x_1,...,x_n)
- Record(T_1,...,T_n)
- TrailingArray((U_1,...,U_n),(V_1,...,V_m))
- Union(T_1,...,T_n)
- Category
- Join(C_1,...,C_n)
- Boolean
- Literal
- Generator T
- Exit
- Foreign I
- Machine
- Ref T
- "Magic Types"

These types are described in the sections which follow.

14.1 Type

• `Type:` `Type`
"`Type`" is the type of all data type objects, including itself. Sometimes it it not possible to tell whether a value is a type. Unless a value has been explicitly asserted to be a type, then it is not treated as one.

In the example below, the parameter "`t`" of the function "`higher`" is a type in some possible calls but not in others.

```
#include "aldor"

higher(T: Type, f: T->T, t: T): T == f f t;

-- next next 2
n: Integer   == higher(Integer, next, 2);

-- List List Integer
L: PrimitiveType == higher(PrimitiveType, List, Integer);
```

14.2 $(S_1,..,S_n)$->$(T_1,..,T_m)$

• `->:` `(Tuple Type, Tuple Type) -> Type`
"$(S_1,..,S_n)$->$(T_1,..,T_m)$" is the type for functions which have n arguments and produce m results of the given types. The types S_i may be mutually dependent and may have default values. The types T_j may depend on S_i as well as on each other.

These are a few examples of function types:

`I ==> Integer`	A macro used in these examples.
`(I, I) -> ()`	No result — useful for side-effecting functions.
`(I, I) -> I`	One result — most common case.
`(I, I) -> (I, I)`	Two results.
`Tuple I -> Tuple I`	Any number of arguments and any number of results.
`(i: I, n: I) -> I`	The arguments may be passed by keyword.
`(i: I, n: I == 0) -> I`	The arguments may be passed by keyword, and the second one has a default value.
`(I, n: I)-> IntegerMod n`	The return type[1] depends on an argument value.
`(n: I, IntegerMod n)-> I`	One argument type depends on another argument value.

14.3 Tuple T

• `Tuple:` `Type -> Type`
Tuples provide n-ary, homogeneous products. The language allows explicit multiple values of the same type or a "`Cross`" of values of the same type to be converted to a Tuple. For example, `()`, `(1)`, `(1, 2)`, and `(1,3,7,8)` may all be used where a `Tuple(Integer)` is expected. The base Aldor library extends Tuple to provide operations to count the elements or extract particular ones:

• `length`: `Tuple S -> MachineInteger`
• `element`: `(Tuple S, MachineInteger) -> S`
These operations are named differently to the standard "`#`" and "`apply`" to avoid ambiguity between values and singleton tuples. The "`element`" operation uses 1-based indexing. Tuple values are not updatable.

14.4 Cross(T_1,...,T_n)

• `Cross`: `Tuple Type -> Type`
This is the constructor for cross-product types. The language allows explicit multiple values to be converted to a single cross product value and *vice versa*. Here is an example:

```
-- Conversion of multiple values to a cross product:
ij: Cross(Integer, Integer) := (1, 2);

-- Conversion of a cross product to multiple values:
(i, j) := ij;

-- This gives the same result as  n := i + j.
n := + ij;
```

There are no operations for counting or selecting product components, and product values are not updatable. The products may be cartesian or dependent:

```
Cross(Integer, Integer)          -- cartesian product
Cross(n: Integer, IntegerMod(n)) -- dependent product
```

14.5 Enumeration(x_1,...,x_n)

• `Enumeration`: `Tuple Type -> Type`
Enumerations are types consisting of a fixed set of symbolic values.

A list of names enclosed in single quotes is a short-hand for a call to `Enumeration`. For example,

```
Colour == 'red, green, blue';

x: Colour := red
```

One of the common uses of enumerations is for selector functions. The `List(S)` domain from the standard Aldor library exports functions called `first`, `rest`, `setFirst!` and `setRest!`. These operations could be replaced by the following ones:

```
apply: (%, 'first') -> S
set!:  (%, 'first', S) -> S
apply: (%, 'rest') -> %
set!:  (%, 'rest', %) -> %
```

This allows expressions of the form:

```
l.first; l.first := s
l.rest;  l.rest  := l
```

Separate types `'first'` and `'rest'` are used, rather than one `'first, rest'`, to allow strong type checking.

The precise way in which enumerations work may seem a bit strange at first: the form

```
'red, green, blue'
```

is actually a short-hand for the call

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

Notice that this has the same form as a typical call to `Record` or a call to "`->`" with keyword arguments. This provides enumerations without introducing any extra fundamental ideas into the language.

14.6 Record(T_1,...,T_n)

- `Record:` `Tuple Type -> Type`

Records provide the basic updatable structure for aggregate data. Each type argument to `Record` may be given in any of the following forms:

T or $id : T$ or $id : T$ `==` v.

A record type `Record(`T_1`,...,`T_n`)` exports the following operations:

- `bracket:` (T_1,...,T_n) `-> %`
- `record:` (T_1,...,T_n) `-> %`
- `explode:` `% ->` (T_1,...,T_n)
- `dispose!:` `% -> ()`

and, for each argument of the form $id : T$ or $id : T$ `==` v, the record type also exports the operations:

- apply: `(%, 'id`$_i$`')` -> T_i
- set!: `(%, 'id`$_i$`', `T_i`)` -> T_i

The "bracket" and "record" operations have the same function and construct new record values.

The "bracket" operation allows records to be constructed with the syntax [1,1], which is nice and concise. More importantly, it allows record types to be used generically — that is, without disclosing that the heterogeneous aggregate is a record type, as opposed to a table or other structure.

The "record" operation allows documented construction of record values. This is convenient if there are many aggregate types in scope (lists, lists of records, records of lists *etc.*) and the bracket operation becomes too heavily overloaded for clarity.

The "explode" operation allows record values to be deconstructed into their constituent parts. This is convenient for use with multiple assignment or passing all the components to a function of an equal number of arguments.

The "dispose!" operation promises that its argument will no longer be referenced, and on certain platforms permits the memory to be reused immediately. It is not necessary to use this function: if you don't, storage will be garbage collected periodically. The choice often boils down to debugability *versus* speed.

The "apply" and "set!" operations allow the record fields to be extracted and reset. This may be done with either explicit calls to these functions or implicit calls arising from the forms "r.i" or "r.i := j".

The substitution of the type arguments into the exported operations uses the original form in which the argument is given. As a consequence, record constructors support keyword arguments and arguments with default values.

To be concrete, we give a few examples.

Record(I, DF) The simplest (and least useful) way to use Record is to call it with simple type expressions, giving neither field names nor default values with the arguments.

An example would be Record(Integer, DoubleFloat). In this case no "apply" or "set!" operations are exported, and the behaviour of the record type is very similar to that of a corresponding call to Cross.

- bracket: (Integer, DoubleFloat) -> %
- record: (Integer, DoubleFloat) -> %
- explode: % -> (Integer, DoubleFloat)
- dispose!: % -> ()

`Record(i:I, x:DF)`

The call `Record(i: Integer, x: DoubleFloat)` provides a record type with the following operations:

- `bracket: (i: Integer, x: DoubleFloat) -> %`
- `record: (i: Integer, x: DoubleFloat) -> %`
- `explode: % -> (Integer, DoubleFloat)`
- `dispose!: % -> ()`
- `apply: (%, 'i') -> Integer`
- `apply: (%, 'x') -> DoubleFloat`
- `set!: (%, 'i', Integer) -> Integer`
- `set!: (%, 'x', DoubleFloat) -> DoubleFloat`

This allows expressions of the form

```
r := [3, 4.0];
r := [i == 3, x == 4.0]
r := [x == 4.0, i == 3]

(vi, vx) := explode r

r.i := 7
r.x := 32.0
```

To be painstakingly correct, the exports are not precisely as shown above. The actual exports are:

- `bracket: (i: Integer, x: DoubleFloat) -> %`
- `record: (i: Integer, x: DoubleFloat) -> %`
- `explode: % -> (i: Integer, x: DoubleFloat)`
- `dispose!: % -> ()`
- `apply: (%, 'i') -> (i: Integer)`
- `apply: (%, 'x') -> (x: DoubleFloat)`
- `set!: (%, 'i', i: Integer)) -> (i: Integer)`
- `set!: (%, 'x', x: DoubleFloat) -> (x: DoubleFloat)`

What is happening is that the exports from the record type all support keyword arguments as a consequence of uniform substitution. This really only useful for the "`bracket`" and "`record`" operations, but it is cleaner to be completely uniform.

`Record(i:I==7, x:DF==0)`

If argument types to `Record` are given with default values, for example `Record(i: Integer == 7, x: DoubleFloat == 0)`, then the uniform substitution yields the following construction operations:

- `bracket: (i: Integer == 7, x: DoubleFloat == 0) -> %`
- `record: (i: Integer == 7, x: DoubleFloat == 0) -> %`

Now it is no longer necessary to supply all the fields when constructing new values:

```
r := [5];      -- Same as  r := [5, 0]
r := [.01];    -- Same as  r := [7, .01]
r := [];       -- Same as  r := [7, 0]
```

14.7 TrailingArray((U_1,...,U_n),(V_1,...,V_m))

- **TrailingArray:** `(Tuple Type,Tuple Type) -> Type`

Trailing arrays are an aggregate data type consisting of two parts; a header, and an array of objects immediately following. The representation for this is a single block of memory, the array immediately following the header. It is up to the user to ensure that accesses to the trailing part of the structure are correct.

For example, a polynomial could be represented as:

`TrailingArray( sz: Integer , (coef: R, deg: NNI))`

This would create a data structure looking like:

sz	coef	deg	coef	deg	coef	deg	...

The advantage of this representation is that the object is created by a single allocation, and that there is no overhead for data pointers (as there would be in a representation using a list of records. For example, the domain

`Record(sz: Integer, tail: PrimArray Record(r: R, deg: NNI))`

contains exactly the same information, but looks like:

$$\begin{array}{cccccccc} \text{sz} & \text{tail} & & & & & & \\ & \downarrow & & & & & & \\ & 0 & & 1 & & 2 & & \ldots \\ & \downarrow & & \downarrow & & \downarrow & & \\ & \text{coef} & \text{deg} & \text{coef} & \text{deg} & \text{coef} & \text{deg} & \ldots \end{array}$$

Usage

`TrailingArray( U, V )`

U and V are tuples of types, and so take the form '(T_1, T_2, ...)', or simply 'T_1' if there is only a single type in the tuple. Assume that U is '(u_1: U_1, u_2: U_2, ...)'

The domain exports the following functions:

The following are equivalent to the exports of `Record(U)` :

- **apply:** (%, 'u_n') -> U_n
- **set!:** (%, 'u_n', U_n) -> U_n

Trailing part access:

- **apply:** (%, MachineInteger, 'v_n') -> V_n
- **set!:** (%, MachineInteger, 'v_n', V_n) -> V_n

Allocation and disposal:

- **bracket:** `(MachineInteger, Cross U, Cross V) -> %`

- `trailing:` `(MachineInteger, Cross U, Cross V) -> %`
- `dispose!:` `% -> ()`

The allocation functions take 3 arguments: a size, and an initial value for both the header and trailing parts. The trailing part argument is ignored at the moment. These have to be filled in by the user, and the initial value of the trailing parts is undefined.

Currently the datatype does not support dependent types at all.

14.8 Union(T_1,...,T_n)

- `Union:` `Tuple Type -> Type`

The `Union` constructor provides types which can be used to represent values belonging to any one of several alternative types.

If a function were to return either an integer or a floating point number, then a type such as `Union(int: Integer, flo: DoubleFloat)` could be used. This type would then provide the following operations:

- `bracket:` `(int: Integer) -> %`
- `bracket:` `(flo: DoubleFloat) -> %`
- `union:` `(int: Integer) -> %`
- `union:` `(flo: DoubleFloat) -> %`
- `case:` `(%, 'int') -> Boolean`
- `case:` `(%, 'flo') -> Boolean`
- `apply:` `(%, 'int') -> Integer`
- `apply:` `(%, 'flo') -> DoubleFloat`
- `set!:` `(%, 'int', Integer) -> Integer`
- `set!:` `(%, 'flo', DoubleFloat) -> DoubleFloat`
- `dispose!:` `% -> ()`

That is, for each argument, $id : T$, the union type exports the following operations:

- `bracket:` `(id: T) -> %`
- `union:` `(id: T) -> %`
- `case:` `(%, 'id') -> Boolean`
- `apply:` `(%, 'id') -> T`
- `set!:` `(%, 'id', T) -> %`

Just as with record types, constructors are available both with a generic name ("`bracket`") and a type-specific name ("`union`"). These form union values from values belonging to the branch types.

The "`case`" operation tests whether the union value is in the given branch. The "`apply`" operation extracts the value and the "`set!`" operation modifies the value of an existing union.

Example:

```
import from Union(num: Integer, rec: Record(c: Character, s: String));

-- Generic constructors
u := [3];
u := [[char "c", "hello"]];

-- Specially named constructors
u := union 3;
u := union record(char "c", "hello");

-- Construction using field names
u := [num == 3]
u := union(rec == record(char "c", "bye"));

-- Testing the case of a union.
u case num;   -- This returns false now.
u case rec;   -- This returns true now.

-- Access the union value as a record.
h := u.rec.s;
```

Using the constructors with keyword arguments is particularly useful when more than one branch of a union has the same type:

```
MyType(R: Type, E: Type): with {
        fun: Union(coef: R, expon: E) -> %;
} == ...

-- This import causes both union branches to be 'Integer'.
import from MyType(Integer, Integer);

fun [coef  == 7];
fun [expon == 7];
```

14.9 Category

• `Category:` `Type`

Often it is desired to work with some specialised collection of types. Each subtype of `Type` in Aldor is a value which, itself, belongs to the type "`Category`". Categories allow parameterised constructions to specify the requirements on type-valued parameters. A type satisfies a category if it provides all the required exports.

The keyword "`with`" is used to form basic `Category` values and it is possible to test whether a type satisfies a category at evaluation time, using "`has`". For more discussion on categories, see Section 7.9.

14.10 Join(C_1,...,C_n)

• `Join:` `Tuple Category -> Category`
The type "`Join(`C_1`,...,`C_n`)`" is a category which has the union of all the exports of the argument categories C_1, ... , C_n. Conditional exports have their conditions `or`ed.

14.11 Boolean

• `Boolean:` `Type`
The type "`Boolean`" is used for the logical values "`true`" and "`false`". Values of this type are expected in the conditions of "`if`" and "`while`", and in other forms. If a value in one of these contexts is not of type "`Boolean`", then an implicit call is made to a `Boolean`-producing function called "`test`". See chapter 12. The base Aldor library extends the basic `Boolean` type.

14.12 Literal

• `Literal:` `Type`
The type "`Literal`" is used to pass the textual form of literal constants as arguments to the constant forming operations "`integer`", "`float`" and "`string`". The result is in a form suitable for use with the exported conversion operations from the "`Machine`" package. This allows constant forming functions to be evaluated at compile time. See chapter 12.

14.13 Generator T

• `Generator:` `Type -> Type`
The type `Generator` T is used to provide values of type T serially to a "`for`" iterator. The following program shows a function to consume the values of a generator:

```
consume(gg: Generator T, f: T -> ()): () ==
        for t in gg repeat
                f t;
```

Values of type `Generator` T are formed by "`generate`" expressions. T is the unified type of the arguments of the `yield`s within the `generate`. For more discussion on generators, see chapter 9. The base Aldor library provides an extension which allows generators to be formed and manipulated using functions.

14.14 Exit

• `Exit:` `Type`
An expression of type `Exit` does not return to the invoking context, and hence does not produce a local result. Expressions formed with "`break`", "`goto`", "`iterate`", "`never`", "`return`" and "`yield`" all have type `Exit` since they do not return directly to the expression containing them.

The unification of type `Exit` with any other type T is T. This allows a function to end with a `return`, *e.g.*:

```
#include "aldor"
f(x: Integer): Integer == { if x < 0 then x := -x; return x }
```

Variables and defined constants may not have type `Exit`, but functions may have `Exit` as their return type. A function with return type `Exit` promises to never return to the caller.

• `error`: `String -> Exit`

Having `Exit` as the return type allows this function to be used in writing programmer-defined error functions. In the standard Aldor library it is provided by `String`. Note that a call to "`error`" will terminate the program.

Example:

```
#include "aldor"

import from String;

errDenom(): Exit ==
     error "Wrong denominator";

f(n: Integer, d: Integer): Integer == {
        d > 0 => n quo d;
        d < 0 => (-n) quo (-d);
        errDenom();
}
```

Notice again that an expression of type `Exit` may appear in a value context. A consequence of this is that expressions such as the following are completely legitimate, but odd:

```
#include "aldor"

l: List Integer := [2, 3, 4, if n < 10 then 5 else error "Too big"];

f(): Integer == return { return { return 7 } }
```

14.15 Foreign *I*

• `Foreign`: `Type`

• `Foreign`: `Type -> Type`

The type "`Foreign`" allows programs to receive values from or provide values to programs which are not written in Aldor. The result of the function "`Foreign`" is similar, but uses an interface for a particular language or environment.

To refer to values which are not produced by an Aldor program, a qualified "`import`" statement such as the following is used:

```
import { DSQRT: DoubleFloat -> DoubleFloat } from Foreign Fortran
```

To provide values to another environment, a qualified "`export`" statement is used:

```
export { J0: DoubleFloat -> DoubleFloat } to Foreign C;
```

The environments which are understood, are:

• `Builtin:` `Type`
The type "`Builtin`" can be used as an interface to abstract machine level operations.

The Aldor compiler, for instance, generates calls to functions for some of the run-time support. These functions are, themselves, written in Aldor and made available to the abstract machine with export declarations such as these:

```
export {
        rtCacheMake:  () -> PtrCache;
        rtCacheCheck: (PtrCache, Tuple Ptr) -> (Ptr, Boolean);
        rtCacheAdd:   (PtrCache, Tuple Ptr, Ptr) -> ();
} to Foreign Builtin;
```

• `C:` `Type`
• `C:` `Literal -> Type`
The type "`C`" is used as an interface to functions written in (or call-compatible with) the C programming language. The type "`C` *`filename`*" is used as an interface to C functions or macros provided by a particular header file. For details, see chapter 19.

• `Lisp:` `Type`
The type "`Lisp`" is used as an interface to functions written in Lisp. This is most useful when Aldor programs are compiled to Lisp code, and allows access to all the operations of the underlying Lisp system (see "`import`" example above).

• `Fortran:` `Type`
The type "`Fortran`" is used as an interface to functions written in Fortran. For details, see chapter 20.

14.16 Machine

• `Machine:` `with ...`
A number of types and operations upon them are provided by "`Machine`". This is the basic vocabulary of hardware-oriented data types out of which all data values in Aldor are composed.

The types exported by `Machine` do not themselves export any operations. Instead, the operations are exported by `Machine` at the same level as the types. This is an example of what is known as a *multi-sorted algebra* in the literature on type systems.

Most programs do not make any reference to "`Machine`" or its exports. In practice, these types and operations are used within low-level libraries to build a set of richer basic types, which themselves provide relevant operations.

The types exported by `Machine` are described below. The exported operations are listed in Section 15.1.

• `XByte, HInt, SInt, BInt:` `Type`
These are integer types of various sizes. The types `XByte`, `HInt`, `SInt` are unsigned byte, half-precision and single-precision integer types, capable of representing values in at least the ranges 0 to 255, -32767 to 32767 and -2147483647 to 2147483647, respectively. The type `BInt` provides a "big" integer type, which, in principle, may be of any size. In practice, the size will be limited by considerations such as the amount of memory available to a program.

• `SFlo, DFlo:` `Type`
These are single-precision and double-precision floating point types.

• `Bool:` `Type`
This type represents the logical values true and false.

• `Char:` `Type`
This type provides characters for natural language text. These should not be used to store what are logically numbers, as the values may be translated from one character set to another in moving data across platform.

• `Nil, Arr, Rec, Ptr:` `Type`
These provide the basis for composite data structures. Values of type `Nil`, `Arr`, and `Rec` may be converted to type `Ptr` as desired.

• `Word:` `Type -> Type`
Values of types `SInt`, `SFlo` and `Ptr` may be converted to this type.

14.17 Ref *T*

• `Ref:` `Type`

This type is mainly used when interfacing Aldor with Fortran, as described in chapter 20. In effect, it allows a reference to an object to be passed as a parameter, rather than a copy of the object itself. This is particularly useful when dealing with Fortran routines since they often update their arguments to return results.

In reality, when calling a Fortran routine with a `Ref(T)` argument, the compiler passes a copy of the initial value to the Fortran code and on exit copies the final value back into the appropriate location (so-called *copy in, copy out* semantics). Not declaring a parameter to a Fortran routine to be a `Ref(T)` is legal even if that routine updates it, it simply means that the new value is not visible to Aldor.

When passing an Aldor routine to a Fortran program, it is necessary to be a bit more careful and use the correct operations to update the values of those parameters declared to be of type `Ref(T)`.

This type has the following exports:

- `ref: T -> %`
- `deref: % -> T`
- `update!: (%,T) -> T`

14.18 Magic Types

In addition to the types defined by the language, there are a small number of domains which need to be present for the interpreter and runtime to function properly. Thus if a user wishes to replace all the standard libraries, they need to provide compatible versions.

The domains are:

- FileName
- Pointer
- MachineInteger
- TextWriter

CHAPTER 15

Standard interfaces

15.1 The machine interface

We provide here a detailed list of the exports from the "`Machine`" package. This package provides the basic machine-level types and operations on them to Aldor.

- `Bool, Char, Nil, Ptr, Arr, Rec, Word:` `Type`
- `XByte, HInt, SInt, BInt, SFlo, DFlo:` `Type`

These are the types provided by "`Machine`". They are described in Section 14.16.

- `true, false:` `Bool`
- `space, tab, newline:` `Char`
- `nil:` `Ptr`
- `0, 1:` `XByte`
- `0, 1:` `SInt`
- `0, 1:` `HInt`
- `0, 1:` `BInt`
- `0, 1:` `SFlo`
- `0, 1:` `DFlo`
- `min, max:` `Char`
- `min, max:` `XByte`
- `min, max:` `HInt`
- `min, max:` `SInt`
- `min, max:` `SFlo`
- `min, max:` `DFlo`
- `epsilon:` `SFlo`
- `epsilon:` `DFlo`

These are special values of the various types: "`min`" and "`max`" are the smallest (most negative) and largest (most positive) values belonging to the finite arithmetic types; "`epsilon`" is the smallest number ε such that $1+\varepsilon$ is distinguishable from 1 in floating point arithmetic.

- `nil?:` `Ptr -> Bool`
- `digit?, letter?:` `Char -> Bool`
- `zero?, positive?, negative?:` `SInt -> Bool`
- `zero?, positive?, negative?:` `BInt -> Bool`
- `zero?, positive?, negative?:` `SFlo -> Bool`
- `zero?, positive?, negative?:` `DFlo -> Bool`
- `even?, odd?:` `BInt -> Bool`
- `even?, odd?:` `SInt -> Bool`
- `single?:` `BInt -> Bool`

These operations provide various tests to inquire about values. The "`single?`" function tests whether a `BInt` value can be represented as a member of `SInt`.

- `=, ~=, <, <=:` `(Bool, Bool) -> Bool`
- `=, ~=, <, <=:` `(Char, Char) -> Bool`
- `=, ~=, <, <=:` `(SInt, SInt) -> Bool`
- `=, ~=, <, <=:` `(BInt, BInt) -> Bool`
- `=, ~=, <, <=:` `(SFlo, SFlo) -> Bool`
- `=, ~=, <, <=:` `(DFlo, DFlo) -> Bool`
- `=, ~=:` `(Ptr, Ptr) -> Bool`

These functions provide tests for equality, inequality and order.

- `upper, lower:` `Char -> Char`
- `char:` `SInt -> Char`
- `ord:` `Char -> SInt`

These operations manipulate character data. The "`upper`" and "`lower`" operations change the case of letters, and leave characters which are not letters unchanged. The "`ord`" and "`char`" are inverse operations mapping between character and integer values.

- `+, -, *:` `(SInt, SInt) -> SInt`
- `+, -, *:` `(BInt, BInt) -> BInt`
- `+, -, *:` `(SFlo, SFlo) -> SFlo`
- `+, -, *:` `(DFlo, DFlo) -> DFlo`
- `/:` `(SFlo, SFlo) -> SFlo`
- `/:` `(DFlo, DFlo) -> DFlo`
- `^:` `(BInt, SInt) -> BInt`
- `^:` `(BInt, BInt) -> BInt`

These are the arithmetic operations of addition, subtraction, multiplication, division and exponentiation.

- `_*_+:` `(SInt, SInt, SInt) -> SInt`
- `_*_+:` `(BInt, BInt, BInt) -> BInt`
- `_*_+:` `(SFlo, SFlo, SFlo) -> SFlo`
- `_*_+:` `(DFlo, DFlo, DFlo) -> DFlo`

These are "multiply-add" operations: `_*_+(a,b,c)` is mathematically equivalent to `a*b + c` but may be faster or have less rounding error.

- `quo, rem, mod, gcd:` `(SInt, SInt) -> SInt`
- `quo, rem, mod, gcd:` `(BInt, BInt) -> BInt`

- `divide`: `(SInt, SInt) -> (SInt, SInt)`
- `divide`: `(BInt, BInt) -> (BInt, BInt)`

These are operations related to integer division. The "`divide`" operations return a quotient-remainder pair.

- `mod_+, mod_-, mod_*`: `(SInt, SInt, SInt) -> SInt`
- `mod_*inv`: `(SInt, SInt, SInt, DFlo) -> SInt`

These operations perform arithmetic modulo their third arguments. The "`mod_*inv`" operation is equivalent to "`mod_*`" but expects an accurate floating point approximation to the inverse of the modulus as its last argument.

- `~`: `Bool -> Bool`
- `~`: `SInt -> SInt`
- `/\`: `(Bool, Bool) -> Bool`
- `/\`: `(SInt, SInt) -> SInt`
- `\/`: `(Bool, Bool) -> Bool`
- `\/`: `(SInt, SInt) -> SInt`

These are bit-wise logical operations.

- `length`: `SInt -> SInt`
- `length`: `BInt -> SInt`
- `bit`: `(SInt, SInt) -> Bool`
- `bit`: `(BInt, SInt) -> Bool`
- `shiftUp, shiftDown`: `(SInt, SInt) -> SInt`
- `shiftUp, shiftDown`: `(BInt, SInt) -> BInt`

These functions provide access to the bits in the base-2 representation of integers.

- `next, prev`: `SInt -> SInt`
- `next, prev`: `BInt -> BInt`
- `next, prev`: `SFlo -> SFlo`
- `next, prev`: `DFlo -> DFlo`

These functions provide next and previous values of the given type. For example, "`next 1`" is 2 for `SInt` and `BInt` values, but might be approximately `1.0000000000000002220446` for `DFlo`, depending on the platform.

- `double_*`: `(Word, Word) -> (Word, Word)`
- `doubleDivide`: `(Word, Word, Word) -> (Word, Word)`
- `plusStep`: `(Word, Word, Word) -> (Word, Word)`
- `timesStep`: `(Word, Word, Word, Word) -> (Word, Word)`

These are the basic operations underpinning multi-word arithmetic.The first two functions provide double word integer multiply and divide operations. The double word multiply instruction returns its results as a pair. The double word division accepts the dividend as a pair, passed as the first two arguments, the divisor as the third argument, and returns the quotient and remainder. The "`plusStep`" operation adds two values and a carry to produce a result and a carry. The "`timesStep`" operation computes the product of its first two arguments, adds in the other two, and represents the result as a word pair. This may be summarised as:

```
double_*:  (rH, rL)    = divide(a * b,            wordsize)
plusStep:  (kout,  r)  = divide(a + b + kin,      wordsize)
timesStep: (rHout, rL) = divide(a * b + c + rHin, wordsize)
```

• `assemble:` `(SInt, SInt, Word) -> SFlo`
• `assemble:` `(SInt, SInt, Word, Word) -> DFlo`
• `dissemble:` `SFlo -> (SInt, SInt, Word)`
• `dissemble:` `DFlo -> (SInt, SInt, Word, Word)`
These operations allow manipulation of the sign $s = \pm 1$, exponent e, and mantissa x of floating point numbers, $f = s \times x \times 2^e$.

• `nearest, zero, up, down, any:` `() -> SInt`
• `round:` `SFlo -> BInt`
• `round:` `DFlo -> BInt`
• `round_+, round_-, round_*, round_/:` `(SFlo, SFlo, SInt) -> SFlo`
• `round_+, round_-, round_*, round_/:` `(DFlo, DFlo, SInt) -> DFlo`
These provide rounded floating point arithmetic.The third argument of the `round_`*op* functions is a rounding mode, as supplied by `Bnearest`, `Bzero` (toward zero), `Bup` (toward plus infinity), `Bdown` (toward minus infinity), or `Bany` (doesn't matter).

• `convert:` `SFlo -> DFlo`
• `convert:` `DFlo -> SFlo`
• `convert:` `XByte -> SInt`
• `convert:` `SInt -> XByte`
• `convert:` `HInt -> SInt`
• `convert:` `SInt -> HInt`
• `convert:` `SInt -> BInt`
• `convert:` `BInt -> SInt`
• `convert:` `SInt -> SFlo`
• `convert:` `SInt -> DFlo`
• `convert:` `BInt -> SFlo`
• `convert:` `BInt -> DFlo`
• `convert:` `Ptr -> SInt`
• `convert:` `SInt -> Ptr`
• `convert:` `Arr -> SFlo`
• `convert:` `Arr -> DFlo`
• `convert:` `Arr -> SInt`
• `convert:` `Arr -> BInt`
These functions convert values between the various `Machine` types. Most of the operations may be accomplished by simply padding or narrowing the internal representation of values. Others such as the integer to floating point conversion are more complex.

• `format:` `(SInt, Arr, SInt) -> SInt`
• `format:` `(BInt, Arr, SInt) -> SInt`
• `format:` `(SFlo, Arr, SInt) -> SInt`

- `format: (DFlo, Arr, SInt) -> SInt`
- `scan: (Arr, SInt) -> (SFlo, SInt)`
- `scan: (Arr, SInt) -> (DFlo, SInt)`
- `scan: (Arr, SInt) -> (SInt, SInt)`
- `scan: (Arr, SInt) -> (BInt, SInt)`

These operations allow the conversion of numeric data to and from a textual form.

- `array: (E: Type) -> (E, SInt) -> Arr`
- `get: (E: Type) -> (Arr, SInt) -> E`
- `set!: (E: Type) -> (Arr, SInt, E) -> E`

These operations allow the implementation of array types.

- `dispose!: Arr -> ()`
- `dispose!: BInt -> ()`

Calling one of these operations indicates that the storage for the argument will no longer be used. On some platforms this makes the storage available for immediate re-use, while on others it does not become available until after the next garbage collection. If a program does not make calls to "`dispose!`", then the storage is reclaimed by garbage collection.

- `RTE: () -> SInt`
- `OS: () -> SInt`

These operations allow programs to inquire about the platform on which they are running.

The "`RTE`" operation indicates the run time environment. This is sometimes useful in selecting between alternative foreign imports. The following values may be returned:

0	The environment cannot be determined.
1	The program is linked as a C-based application.
2	The program is running on top of a Common Lisp system.

The "`OS`" operation specifies the operating system. This is sometimes useful in understanding how to parse file names, for example. The values below correspond to the given operating systems:

OS() quo 1000	Operating system
0	Cannot be determined.
1	A Unix derivative.
2	IBM VM/CMS.
3	OS/2.
4	DOS for IBM PC compatibles.
5	Microsoft Windows.
6	DEC VMS.
7	Macintosh System 7

- `halt: SInt -> Exit`

Finally, this operation terminates the execution of a program. On platforms which support it, the integer argument is returned to the calling environment.

15.2 Standard libraries

Programs will normally use more than just the language-defined types. The example programs in this guide make use of the standard Aldor library, also called `libaldor`.

The standard Aldor library provides a set of basic types for numbers, data structures, objects, input, output and so on. This library should be used when developing stand-alone programs, or programs to link with C- or Fortran-based applications.

The simplest way to make the base Aldor library available within an Aldor is to use a line of the form

```
#include "aldor"
```

This incorporates the text of the standard header file "`aldor.as`" into the program being compiled.

The standard "`aldorio.as`" can also be included:

```
#include "aldor"
#include "aldorio"
```

While "`aldor.as`" makes the library visible, this extra include file also defines a few standard macros and imports a few basic operations from types in the library. So, for example, after including these files, "`stdout`" and "`<<`" have meanings.

This header file uses a `#library` command to make the library archive "`libaldor.al`" available as a package called "`aldorlib`" within the program. Normally it is not necessary to use the name "`aldorlib`" at all, but if you are using other libraries which introduce colliding exports, you can disambiguate the collisions by referring to the base library entities with a `$`-qualification, such as "`Integer$aldorlib`" or "`+$Integer$aldorlib`".

PART III

The Aldor compiler

CHAPTER 16

Understanding messages

The Aldor compiler may display messages of various sorts, including errors, warnings and remarks about the program being compiled. This section is a guide to understanding the format and controlling the generation of Aldor messages.

A complete list of the messages produced by the compiler may be found in chapter 25.

16.1 Aldor error messages

Even a particularly skilled programmer will have to deal with compiler error messages at some point. Upon initial encounter, these messages often seem confusing. However, with practice their meaning will become clear. The Aldor compiler tries to give messages which are reasonably brief, but still informative. Although more detailed messages may be useful to novice programmers, most beginners quickly move past the initial desire for more detailed messages. The following paragraphs describe the components of the messages produced by the Aldor compiler.

Format of Aldor error messages

Each message produced by the Aldor compiler is assigned a line and a column position based on the place in the source text of the program associated with the message. Then, for each source line for which messages have been generated, the compiler reports the source line, a line containing pointers to the column assigned to each message, and a description of each message which appears for the source line.

Each source line is reported by showing the source file name in which the line appears (enclosed in quotation marks), the line number within the file, followed by a colon (`:`), and then the text of the line itself.

After the source line is reported, the next output line contains a circumflex (`^`), used as a caret below each column at which an error was detected. Dots (`...`) are used to fill in the spaces between the carets.

Each error that appears on a line is then precisely located and shown in a format which can be used to quickly locate and fix the error. For each error, the line and column number are printed first, enclosed within square brackets (`[ ]`), then the severity of the error is indicated by one of the keywords "`Fatal`", "`Error`", "`Warning`", or "`Remark`" appearing in parentheses. Following the severity of the error is the text of the message, which may include references to parts of the source text, or references to types or other data structures used to compile the program.

16.2 Example showing Aldor messages

The message format described in the previous section will be illustrated by examining the error messages produced when we attempt to compile the file "`error0.as`" shown in Figure 16.2. The command "`aldor error0.as`" produces the messages shown in Figure 16.2.

The first line of this message indicates that the message was generated from line 4 of the file "`error0.as`". The next lines specify that errors were detected at columns 12 and 15 on this line. The carets indicate that the first error reported on this line is due to the operator `=>` and that the second error reported on this line resulted from the symbol `1`. In addition, these are the second and third messages, respectively, which were encountered during the compilation of "`error0.as`".

The next message shown is actually the first detected by the compiler. It is a "Warning" which complains of a missing "`;`". There is also a hint which suggests that piling has been used incorrectly. However, since the declaration "`#pile`" was not included in the file, piling syntax (see Section 22.3) is not being used.

Let's add the declaration "`#pile`" to "`error0.as`" and call the updated file "`error1.as`", shown in Figure 16.2. Compiling this file produces the messages shown in Figure 16.2. Now there is a single error message at line 6, which complains that no meaning can be found for the operator "`f`". There is certainly no declaration for "`f`" in "`error1.as`". This operator should in fact be "`factorial`", and not "`f`". After making this change (shown in Figure 16.2), the file will compile cleanly.

This example was compiled using the default message options as described in Figure 16.5. The compiler's message options determine what kinds of messages the compiler will generate.

```
#include "aldor"

factorial(n: AldorInteger): AldorInteger ==
    n <= 1 => 1
    n * f(n-1)
```

Figure 16.1: "error0.as" — A program containing mistakes.

Figure 16.2: Error messages for "error0.as".

```
#include "aldor"
#pile

factorial(n: Integer): Integer ==
    n <= 1 => 1
    n * f(n-1)
```

Figure 16.3: "error1.as" – A program containing fewer mistakes.

Figure 16.4: Error messages for "error1.as".

```
#include "aldor"
#pile

factorial(n: Integer): Integer ==
        n <= 1 => 1
        n * factorial(n-1)
```

Figure 16.5: "error2.as" – A program which compiles.

16.3 Some common error messages

This section explains some error messages likely to be seen by new users.

Compiler error messages do not always explain the real nature of the problem. This statement is true in general, and not only for the Aldor compiler. Compilers analyse programs on the basis of more or less rigid syntactic rules, and they can become enormously confused by a missing character, or misspelled word.

What follows is a collection of 'mysterious' messages (or, at least, messages which are puzzling to new Aldor programmers), generated by Aldor.

• 'aldor.as' cannot be opened

```
"myfile.as", line 1: #include "aldor.as"
                     ^
[L1 C1] #1 (Error) Could not open file 'aldor.as'.

"myfile.as", line 1: import from Integer;
                     ............^
[L1 C13] #1 (Error) No meaning for identifier 'Integer'.
```

This is probably a message that will occur only once — it indicates that Aldor has not been set up properly.

- Check the `ALDORROOT` environment variable. For Unix users, for example, the file "`aldor.as`" must be located in the directory "`$ALDORROOT/include`"; for DOS users it must be located in the directory "`%ALDORROOT%\include`". (Type "`set`" on DOS to see the value of this environment variable. Check the installation instructions to ensure that you have completed all necessary steps.)
- (For DOS users only) check "`FILES=`..." in your `config.sys`. A minimum value of 40 is recommended.

See also Section 23.14.

• No meaning for integer-style literal '*xxx*'
• No meaning for string-style literal '*xxx*'
• No meaning for float-style literal '*xxx.xx*'

Unlike traditional programming languages, in Aldor there is no built-in knowledge about the meaning of numeric or string constants. Every domain can export an interpretation for these constants, with the advantage that you can define your own methods for interpreting them. The standard Aldor library contains some standard domains exporting interpretations for these constants, but you must explicitly import these domains in the scope of your program. So in your program you need to say:

`import from MachineInteger;`   *or*   `import from Integer;`

to specify if you want to use the machine representation or the infinite precision representation for integer-style literals such as 2, 3, 4, ...

For string-style literals, such as "`dog`" and "`cat`", you can import from the "`String`" domain, and for floating point-style literals you can import from "`SingleFloat`" (single precision hardware floating point) or "`DoubleFloat`" (double precision hardware floating point). See also Section 5.2.

- **No one possible return type satisfies the context type**

Aldor is a strongly typed language. This means that the type of each expression forming the program is determined during the compilation process. The following example shows a typical situation in which this message occurs:

```
#include "aldor"

foo():Integer == 1;

b: MachineInteger == foo();
```

Compiling this example with the option "`-M2`" (full error messages), gives:

```
"myfile.as", line 5: b: MachineInteger == foo();
                     .....................^
[L5 C22] #1 (Error) No one possible return type satisfies the context type.
  These possible return types were rejected:
          -- AldorInteger
  The context requires an expression of type MachineInteger.
```

"`b`" is a constant whose type is "`MachineInteger`", so on the right-hand side of the definition an expression of the same type is required. When the compiler searches for all the possible return types of the expression "`foo()`", it finds that the unique possible type, "`AldorInteger`" does not satisfy "`MachineInteger`"[1]. Note that a function could be overloaded, which is why the message says: "`No one possible return type`...". To fix this problem, "`foo`" can be overloaded with a new definition, or the result of foo can be *coerced* to a `MachineInteger` as in:

```
b: MachineInteger == foo()::MachineInteger;
```

See also part II.

- **No meaning for identifier '*domain*'**
  - If you are defining *domain* in the same file, are you sure that the spelling for *domain* is correct? (Remember that Aldor is case-sensitive: "`Integer`" and "`integer`" are two different identifiers!)

[1]The error message is in terms of `AldorInteger`, whereas the source code uses `Integer`. This is because the `aldor.as` file maps the readable name `Integer` to the unique identifier `AldorInteger`, as explained in Section 2.1.

- If *domain* is from an external library, for example "`mylib`", did you include the "`#library`" statement in your file? For instance:

```
#library MyLib "mylib"
import from MyLib;
```

  In this case, if you get the error message in the import statement and the symbol that you are importing has been defined in a "`#library`" statement, then check the following rules:

  1. If the library is a library created with "`ar`" (with suffix .al), than you must *not* write the suffix. This is the case, for "`mylib`" — it corresponds to the file "`libmylib.al`"
  2. If the library is a "`.ao`" file, than you **must** use the `.ao` suffix in the `#library` statement.

See also Section 17.2.

- **Argument 1 of 'test' did not match any possible parameter type**

If the test is for equality, in an "`if`" statement, are you sure that you are using "`=`" and not "`==`"? This can be a frequent mistake for C/C++ programmers. The message says "`Argument 1 of 'test'`..." even if there is no explicit call to the "`test`" operator, because in some contexts, such as an "`if`" statement, the "`test`" operator is implicitly applied.

- **'Syntax error' at the beginning of a line**

If you are sure about the correctness of the displayed line, look at the previous line. A macro definition using "`==>`" and not ending with "`;`" can often cause this error. For example, compiling the following statements from the file "`myfile.as`":

```
#include "aldor"
-- Not using '#pile'.

MI==>MachineInteger

import from MI;
```

gives the error message:

```
"myfile.as", line 6: import from MI;
                     ^
[L6 C1] #1 (Error) Syntax error.
```

See also chapter 22.

16.4 Common pitfalls

This is a collection of mistakes frequently made by novice Aldor programmers. A subclass of this consists of common mistakes for programmers from other languages, such as C, C++ and Fortran. It is important to be able to recognise situations where such mistakes occur, so that you can understand the compiler's error messages.

• '==>' and '=>'

These symbols have completely different uses in Aldor. The "`==>`" (long arrow) symbol is used to define a macro (you can also use the form "`macro` ..."), see chapter 13, whereas the "`=>`" (short arrow) symbol is used as an early exit in a sequence, see Section 5.10.

• '=' and '==', '=' and ':='

The "`=`" symbol is usually used as an equality operator (but can be overloaded by user programs). The "`==`" (double-equal) is used to define constants — categories, domains, functions and values are all Aldor constants. The double-equal symbol is sometimes referred to as the "very-equal" symbol for obvious reasons. The "`:=`" operator is used for assignments.

• Using '_' as a separator in identifiers

The "_" (underscore) symbol is used as an escape character in Aldor — thus the identifiers "`list_of_integers`" and "`listofintegers`" are exactly the same. For this reason, to prevent ambiguities, we suggest using capital letters, instead of "_", to separate words in identifiers, as in "`listOfIntegers`".

• Missing ';' at the end of a macro definition

Unlike other languages, such as C, the Aldor preprocessor requires that macro definitions end with a "`;`" (or a newline, if you are using "`#pile`". Unfortunately the compiler has no way of knowing that the statement defining the macro has ended and so will usually complain about a syntax error at the start of the next line.

• Undefined symbol _INIT__xxx referenced from text segment

This is a message generated by the C compiler (cc, gcc, xlc, *etc.*) that Aldor is using on your platform to generate stand-alone executable files. Everything in the compilation process of the Aldor program succeeded, but something went wrong in the final linking step.

One common cause of this kind of problem is compilation using another library, as well as/instead of the standard Aldor library `libaldor`, without letting the compiler know about the other library. When you include a "`.al`" library, you should add `-l`*libname* to the command line[2].

See chapter 17 and, in particular, Section 17.2 to understand how to use other libraries with Aldor. A particular point to note when *compiling* is:

- If you want to use a domain from "`mylib`" (a library containing

[2]Note that Aldor options, such as `-l`, are case-insensitive.

definitions of some domains), you should include the following lines in your Aldor program:

```
#library MyLib "mylib"
import from MyLib;
```

(Note that any identifier may be used in place of "MyLib".) When calling the compiler, you need to include the "-lmylib" option among the parameters; for example:

```
% aldor -lmylib myprog.as
```

- **User program with the same name as a library file**

If you create a ".as" file with the same name of a library file, it won't compile. The compiler will issue a warning message if there is such a name conflict. For example, there is a file in libaldor with the name "sal_int.as"; suppose that your file is also named "sal_int.as", when your file is compiled you will get a warning, followed by one or more error messages relating to the symbols being used in your file:

```
#1 (Warning) Current file over-rides existing library in
 '/usr/local/aldor/lib/libaldor.al'.
"pointer.as", line 4: foo():Integer == 1;
                      ................^
[L4 C18] #2 (Error) No meaning for identifier '1'.
```

The easiest way to view the names of the files in a library is usually to use a command provided for that purpose by your operating system — for example, the "ar" command in Unix (see Section 17.2 for an alternative):

```
cd $ALDORROOT/lib
ar vt libaldor.al
ar vt libfoam.al
```

The compiler will issue a warning message if the name of your file is used by any library needed for compilation of the file. If you are recompiling a library, then the warning can safely be ignored.

16.5 Controlling compiler messages

The types of messages returned by the Aldor compiler are controlled using the -M option, which takes an argument identifying the kind of message information to display. Preceding any message option with "no-" negates that option, so -M no-details will suppress the details associated with a compiler message. A complete listing of the message options can be found in Section 23.12.

The default compiler message options

The default message option used by the Aldor compiler is -M 2, which is equivalent to the following collection of options: -M number, -M sort, -M warnings, -M source, -M details, -M notes, -M mactext, -M abbrev and -M human.

The option -M human indicates that the compiler messages are to be read and interpreted by a human user, as opposed to a computer program.

The `-M number` option enumerates the order in which the compiler messages were encountered. An ordinal `#`n appears after the bracketed line and column values. The default option `-M sort` shows the compiler messages sorted by the order in which they appear within the file. Using `-M no-sort` instead causes the messages to appear in the order in which they are encountered by the compiler, which may be different from their order in in the program.

The `-M warnings` option will display the compile time warnings which do not prevent a successful compilation. When the option is changed to `-M no-warnings` then those messages, which are only advisory in nature, are suppressed.

The compiler normally displays the source text which caused each message. However, using `-M no-source` displays only the compiler message with the line and character location without showing the source text.

Some compiler messages may have notes associated with them which cross reference other compiler messages having a similar or identical meaning. They are produced by the option `-M notes`.

Compiler messages abbreviate the data types which are contained within the text of the message, when the option `-M abbrev` is used. To expand the type names, use the option `-M no-abbrev`.

16.6 Interactive error investigation

In most cases, the error messages will give enough information for the programmer to determine what is wrong with a program. In some cases, however, extra detail is required. Rather than print many lines of debugging information for each error, the Aldor compiler gives messages of moderate length and allows the programmer to get more information interactively.

The `-M inspect` option (not to be confused with `-G loop`) must be given if interactive error investigation is desired. Then, whenever the compiler detects an error, it stops and enters a debugger or "break loop". The break loop gives the programmer access to information such as the types of variables and the names in scope. An example using the interactive loop is shown in Figure 16.6.

```
% aldor -Minspect error3.as
"error3.as", line 33:
        b := left(b, z)                 -- lbi-b+zi or lbb-b+zb: ambiguous
............^
[L33 C14] #1 (Error) There are 2 meanings for the operator `left'.
        Meaning 1: (Boolean, Boolean) -> Boolean
        Meaning 2: (Boolean, AldorInteger) -> Boolean

"error3.as", line 34:         b := left(i, i)                 -- no meaning
                      ..................^
[L34 C19] #2 (Error) Argument 1 of `left' did not match any possible parameter type.
    The rejected type is AldorInteger.
    Expected type Boolean.

Aldor compiler break -------------------------------------------------------
::: up
left(b, z)
::: down
left
::: next
b
::: means
1: Param b : Boolean

::: next
z
::: means
1: LexConst z : AldorInteger
2: LexConst z : Boolean

::: quit
-------------------------------------------------------------------------------
```

Figure 16.6: Interactive Error Investigation

The interactive error inspector allows the following commands:

help	list the commands, with descriptions
show	show the current node
means	show the possible meanings of the current node
use	show how the current node is used
seman	show the extra semantic information for the current node
scope	show information about the current scope
up	use the parent as the current node
down	use the 0th child as the current node
next	use the next sibling as the current node
prev	use the previous sibling as the current node
home	return to the original node
where	return the source position of the current node
getcomsg	get information on the current message
notes	show the notes associated with the current message
mselect *i*	select message *i* to be the current message
mnext	select the next message in the list
mprev	select the previous message in the list
msg	display the error message again
nice	show with pretty printed form
ugly	show with more detailed, internal form
quit	exit the compiler, showing all messages so far

16.7 Selecting error messages

The Aldor compiler is organised in such a way that messages may be replaced, enabled, and/or disabled from the command line. Messages are identified by a name tag, *e.g.* "AXL_W_CantUseLibrary". The name tags of all messages used by the compiler may be found in the catalogue of error messages in chapter 25.

Once the name of a particular message is identified, the command

`aldor -M` *msgname* `input.as`

can be used to enable the message *msgname* if it is otherwise disabled. Conversely, the command

`aldor -M no-`*msgname* `input.as`

will disable the message *msgname* if it is otherwise enabled. Normally, the messages classified as warnings are enabled, while those classified as remarks are disabled.

The special message name "`warnings`" can be used to enable or disable all warning messages, and the special name "`remarks`" can be used to enable or disable all remarks. "`Error`" messages cannot be disabled.

In addition to selecting error messages by name, it is possible to use the command

`aldor -M emax=`n

to specify the number of error messages which will be reported before `aldor` gives up and stops processing the input file. The default is ten.

16.8 Error messages and macros

When an error occurs in the processing of program text which involves a macro, it is sometimes useful for the error message to point to the macro invocation, and sometimes for it to point into the body of the macro to the location where the error was detected. The command

```
aldor -M mactext
```

instructs `aldor` to point to the text of the macro, while the command

```
aldor -M no-mactext
```

instructs `aldor` to point to the macro invocation.

There is no mechanism for instructing `aldor` to point to the invocation of a macro which appears in the body of another macro.

16.9 Error messages and GNU Emacs

GNU Emacs offers several commands for compiling and debugging programs. Among these are "`M-x compile`", which can be used to compile Aldor programs. By default, `M-x compile` issues the command "`make`", in a separate process, but the command may be changed to any other Unix command, including any command line which invokes the Aldor compiler. Output from the command goes to the buffer `*compilation*`. The format of the error messages produced by the Aldor compiler is then understood by the command "`` C-x ` ``", which finds the next error message in the `*compilation*` buffer and displays the source line which produced the error.

16.10 Using an alternative message database

It is possible to have the Aldor compiler use an alternative message database. This is done via the `-Mdb` command line option. Thus, a command script can cause the compiler to use a database of messages translated to some other language, or which give a different level of detail.

The messages which have been built into the compiler were derived from the database `samples/comsgdb.msg`. A replacement message database must provide messages for all the message tags in that file, and should be in the X/Open format[3].

Once a new message database file has been created, it can be placed in any directory which is normally searched for executable programs. If the database is called `newcomsgs.cat,` for example, then the compiler argument "`-Mdb=newcomsgs`" will find it.

[3] Note that the `comsgdb.msg` file is *not* in that format

CHAPTER 17

Separate compilation

This chapter describes how to run the Aldor compiler to produce libraries or executable programs from multiple files.

17.1 Multiple files

Unless a program is very small, it is normal to develop it in stages and to identify parts of it for potential re-use, compiling them separately. Aldor supports separate compilation to platform-independent "`.ao`" files. These files contain type information, intermediate code and other information to allow type-safe separate compilation and cross-file optimisation. These files have a portable format and it is possible to move them between machines of different architectures, with different character sets, byte orders or floating-point formats. The "`.ao`" files can be thought of as platform-independent object files, which may be imported into other Aldor programs or used to generate C or Lisp or object code for a particular platform.

Let us give a a toy example which illustrates the steps one takes to do separate compilation with the Aldor compiler. Suppose we have two files: "`choose.as`", containing some functions, and "`poker.as`", which uses them. These files are shown in Figure 17.1. The commands to compile these individual files together are:

```
% aldor -O -Fao -Fo choose.as
% aldor -O -Fao -Fo -lChooseLib=choose.ao  poker.as
% aldor -e poker -Fx -laldor poker.o choose.o
```

The first step compiles the file "`choose.as`". The "`-Fao`" option causes the compiler to create a platform-independent file, "`choose.ao`", and the "`-Fo`" option gives a platform-dependent object file, such as "`choose.o`" on Unix, containing machine code. The "`-O`" option requests optimised code.

```
--
-- choose.as: A file providing functions to be used elsewhere.
--
#include "aldor"

factorial(n: Integer): Integer ==
        if n <= 2 then n else n * factorial(n-1);

choose(n: Integer, k: Integer): Integer ==
        factorial(n) quo (factorial(k) * factorial(n-k));
```

```
--
-- poker.as: A main program using functions from "choose.as".
--
#include "aldor"
#include "aldorio"

import from ChooseLib;
import from Integer;

pok := choose(52, 5);
stdout << "The number of different poker hands is " << pok << newline;
```

Figure 17.1: A program consisting of two files.

The second step compiles the file "`poker.as`", which uses "`choose.as`". The "`-lChooseLib=choose.ao`" option tells the compiler that the library "`choose.ao`" is to be made visible within the program as a package named "`ChooseLib`". This "`-l`" option could be avoided by using a `#library` command, as described in Section 17.3.

In the third command, the "`-Fx`" option directs the compiler to link an executable image from the object files on the command line. The "`-e`" option causes the resulting program to begin execution by evaluating the top-level statements of the file "`poker.as`". The "`-e`" option also directs the compiler to give the executable file the name "`poker`" (possibly with some system-dependent extension, such as "`.exe`" under DOS).

17.2 Libraries

In order to handle large numbers of separately compiled files, the Aldor compiler allows "`.ao`" files to be combined into aggregate "`.al`" files. The aggregate library files may be created and maintained on Unix using the "`ar`" command. For the platforms for which "`ar`" is not available, the Aldor distribution provides a program "`uniar`".

When specifying a library to the compiler with the `-l`*name* option, *name* is treated as a filename if it has a file extension or a directory specification. Otherwise it is treated as a shorthand reference to either or both of the libraries "`lib`*name*`.al`" and "`lib`*name*`.a`" in the current directory or on `$LIBPATH`.

Examples:

- The command-line argument `-lnewmath` is treated as a reference to the library `libnewmath.al` in the current directory or on `$LIBPATH`.
- The command-line argument `-lmyfile.ao` is treated as a reference to the file `myfile.ao` in the current directory or on `$LIBPATH`.
- The command-line argument `-lstuff/myfile.ao` is treated as a relative pathname reference to the file `myfile.ao` in the directory `stuff`.
- The command-line argument `-l/u/joe/myfile.ao` is treated as an absolute pathname reference to the file `myfile.ao` in the directory `/u/joe`.

When the Aldor compiler is asked to create an executable file, references to libraries via `-l` will be passed to the linker, if appropriate. This can be convenient on certain platforms. For example, on Unix, it allows one to have "`lib`*xxx*`.al`" and "`lib`*xxx*`.a`" containing the platform-independent "`.ao`" files and the platform-dependent "`.o`" files, respectively.

17.3 Source code references to libraries

A separately compiled module can be referenced directly from the source text of a program much the same way that it is referenced from the command line. In a source program, the system command `#library` plays the same rôle as `-l` does from the command-line. The treatment of names in the `#library` system command is exactly the same as in the `-l` command-line argument (see Section 23.5. When using the `#library` system command, however, remember to put the name in quotation marks.

References to files using `#library` will be passed to the linker under the same circumstances as the corresponding `-l` command-line argument.

17.4 Importing from compiled libraries

Compiled object files (`.ao`) and compiled libraries (`.al`) have the same top-level semantics as Aldor domains: before the symbols defined in a separately compiled file are visible during the compilation of a given source file, the symbols must be imported into the current scope. Before a domain can be imported, it must be given a name. The same is true of compiler object files: before symbols can be imported from an object file or a library, it must be given a name.

Names are assigned to compiler libraries using an extension of the `-l` syntax and an extension of the `#library` syntax described in the previous two sections. If the argument to either of these forms is prefixed by a symbol (which should *not* be enclosed in quotation marks), then the compiler object or library which follows is taken as the definition of a domain whose name is the given symbol. The symbols exported from the newly named domain are exactly those exported from the source code used to produce the compiled object.

Consider the following Aldor source file:

```
#library String "/tmp/mystring.ao"
import from String;

...
```

The first two lines specify that the domain `String`, usually provided by the compiler libraries found in the distribution, will, for the extent of the file, be provided by the compiled object file "`/tmp/mystring.ao`". The `#library` command specifies a binding for the symbol `String`, and then the `import` command operates as it does in any other context, extracting the exported symbols from the domain `String` and making them visible in the current scope. Note that, unlike the command line usage, the "`#library`" command requires a space, rather than "`=`", between the domain and library names.

CHAPTER 18

Using Aldor interactively

This chapter describes how to use a built-in interpreter to run Aldor programs interactively. We shall assume that the reader is familiar with at least the basic concepts of the Aldor programming language.

18.1 How to use the interpreter

The interpreter is built into the Aldor compiler. It is used in two different contexts:

- running Aldor programs without compiling them to executable files,
- writing Aldor programs in interactive mode.

18.1.1 Running programs with the interpreter (-g interp)

Suppose you write an Aldor program "`foo.as`". One way to get it running is with the command:

```
% aldor -g run foo.as
```

When you call the compiler with this option, it compiles the program into a device-independent intermediate format called *Foam* (stored, in this case, in the file "`foo.ao`"), which is then translated into C code ("`foo.c`") and compiled with the C compiler available on your platform. Finally, the executable code generated by the C compiler is executed[1]. If you call the compiler with the "`-g interp`" option, e.g,

```
% aldor -g interp foo.as
```

the intermediate file "`foo.ao`" is executed without any call to the C compiler. Since the instructions are interpreted, and not pure machine

[1]Note that using only the option "`-g run`" causes all the intermediate files to be removed, once the program is terminated. If you want to keep them, then you should use, for example, "`-Fao -Fx`". See chapter 23 or use the "`-hf`" option to get help.

code instructions as when using "-g run", execution will generally be slower than in the first case. Despite the low execution speed, there are reasons to use the interpreter instead of the compiler:

- as the generation and the compilation of C code is unnecessary, you can get faster responses using the interpreter if the program is not computationally time-expensive;
- on some platforms, such as Intel 80x86 with Windows, there is no C compiler included with the operating system; in the absence of any C compiler, the interpreter is the only way of running an Aldor program.

Using the interpreter is suggested especially for those who are learning Aldor, as it provides a quick way of testing small programs. Note that all the compiler options can be used with "-g interp". All optimisations except "-Qcc" are still effective, because all optimisations in Aldor are performed on the intermediate code. So, for example, if the compiler is called using the "-Q3" option, as in:

```
% aldor -Q3 -g interp foo.as
```

the program will generally run faster than without the "-Q3" option.

Note that the semantics of the language are fully preserved by the interpreter. The interpreter provides all of the Aldor language features.

18.1.2 Interactive mode (-g loop)

The **interactive mode** provides an interactive environment in which it is possible to define functions and domains, to use operations provided by the library, to evaluate expressions and to use other features. Users who are familiar with programming in languages that are usually interpreted, such as Lisp, already know the feeling of an interactive environment and how it can be used to gain confidence in the language, and to develop and debug more complex programs.

The command line to start the compiler in interactive mode is:

```
% aldor -g loop
```

A prompt will appear:

```
%1 >> _
```

At this point, you start typing the first line of your program, for example:

```
%1 >> #include "aldor"
```

After a few milliseconds the prompt appears again (the delay is due to the interpreter loading the base library):

```
%1 >> #include "aldor"
%2 >> _
```

A nicer output for types and values is available by include "`aldorinterp.as`":

```
%2 >> #include "aldorinterp"
```

The number that appears immediately after `%` is a serial number which is incremented with each input line from the user. As will be seen, this is useful when the `history` mode is `on`. Now type:

```
%3 >> import from MachineInteger
%4 >> 1 + 1
```

At this point the expression `1 + 1` will be evaluated and the answer is:

```
2 @ MachineInteger
```

in which `2` is the result of the expression and `@ MachineInteger` means that its type is `MachineInteger`. To quit the interactive mode type:

```
%5 >> #quit
```

When the interpreter starts, it looks in the current directory (the directory from which the command line is being entered) for an initialisation file. An initialisation file is any Aldor program named `aldorinit.as`. If this file is present in the current directory, it will be loaded and executed before the prompt appears.

An example initialisation file could be:

```
#include "aldor"
#include "aldorinterp"

-- Commonly used macros
MI ==> MachineInteger
I  ==> Integer
```

Note: the interpreter will display a message: `Reading aldorinit.as...` if an initialisation file is read.

The majority of command line options are still active when "`-g loop`" is used. For example, the optimisation option:

```
% aldor -Q3 -g loop
```

will invoke the optimiser before interpreting the generated Foam intermediate code, thereby affecting the execution speed.

We will now try a simple interactive session. Note that all the lines which do not start with the prompt have not been typed, but are part of the output.

```
%1 >> -- Example of interactive session --
%2 >> ------------------------------------
%3 >> -- Start by loading definitions
%4 >> #include "aldor"
%5 >> #include "aldorinterp"
%6 >> import from Integer
%7 >> 100
100 @ AldorInteger
%8 >> 100 + 100
200 @ AldorInteger
%9 >> f(x:Integer):Integer == if x=0 then 1 else x*f(x-1)
Defined f @ (x: AldorInteger) -> AldorInteger
%10 >> f 4
24 @ AldorInteger
%11 >> import from List Integer
%12 >> reverse
  () @ List(AldorInteger) -> List(AldorInteger)
%13 >> reverse [1,2,3,4,5]
[5,4,3,2,1] @ List(AldorInteger)
```

18.2 Directives for the interactive mode

This section provides a full description of some language directives available only in interactive mode[2].

The options available specifically for interactive mode are requested using "`#int`". A brief help message displaying all available options may be obtained with:

`#int help`

The options are as follows:

- help
- verbose
- history
- confirm
- timing
- msg-limit
- gc
- options
- cd
- shell

These are described in detail below.

- `#int verbose [on | off]`

Default is: `on`.

[2] The directives available in interactive mode will be ignored if they are encountered during normal compilation.

When `verbose` is `on`, the interpreter prints, if possible, the value and the type of the current expression.

Example:

```
%4 >> 4
4 @ MachineInteger
%5 >> 5 + 5
10 @ MachineInteger
%6 >> foo(x : String) : Boolean == empty? x
Defined foo @ (x: String) -> Boolean
```

If the value of the expression is not printable (that is, its domain does not export `<<:(%, TextWriter) -> TextWriter`) or if it has no value at all (for instance, in "`import from`" statements), nothing is displayed.

- `#int history [on | off]`

Default is: `off`.

When `history` is `on`, the interpreter wraps, if possible, an assignment around the current expression. If, for example, "`%5`" is the current interpretation step, the prompt will change from

"`%5 >>`"

to

"`%5 :=`"

which means that, if the current expression has a value, this is assigned to a new variable named "`%5`". The variable "`%5`" is implicitly declared and its type is inferred from the type of the right hand side.

Example:

```
%4 >> #int history on        -- history is on
%5 := MI ==> MachineInteger  -- no value is assigned to %5
%6 := import from MI         -- no value is assigned to %6
%7 := 5 + 5                  -- 10 is assigned to %7
10 @ MachineInteger
%8 := 10 + %7                -- you can use %7
20 @ MachineInteger
%9 := %7 := %7 - 5           -- 5 is assigned to %7 and %9
```

Notes:

1. At interpretation step 9, the right association rule for the `:=` operator in Aldor is observed.
2. At steps 5 and 6, since no value is assigned to %5 or %6, the variable names %5 and %6 are *not* introduced and trying to use them would generate an error message.

- `#int confirm [on | off]`

Default is: `on`.

When `confirm` is `on`, the interpreter asks for confirmation before executing some operations that are illegal in the compiled Aldor language. Typical cases are the redefinition of constants and functions: according to the language definition, these cannot be redefined — but it may be useful to relax this rule in interactive mode.

Suppose, for example, that you write a definition for a function "`foo`", later adding other functions using "`foo`"; at some point you want to provide a better implementation for "`foo`": if your program is going to be compiled, then you can simply edit the file where "`foo`" is defined and change it but, unfortunately, this cannot be done if you wrote "`foo`" interactively. This is why, in interactive mode, you can enter a new definition for "`foo`": a message will appear because you are doing something that is not normally allowed in Aldor and, at this point, you can confirm that you want to replace the old definition with the new one. If you do not want the message asking for a confirmation, you can enter:

"`#int confirm off`"

in which case a positive answer is assumed for each situation in which a confirmation would be needed.

Example:

```
%3 >> Int ==> Integer
%4 >> import from Int
%5 >> foo(x : Int) : Int == x
Defined foo @ (x: AldorInteger) -> AldorInteger
%6 >> foo(x : Boolean) : Boolean == not x       -- (see Note 1)
Defined foo @ (x: Boolean) -> Boolean
%7 >> foo(x : Int) : Int == x * x
Redefine ? (y/n): y                             -- (see Note 2)
foo redefined.
Defined foo @ (x: AldorInteger) -> AldorInteger
%8 >> foo 2
4 @ AldorInteger
```

Notes:

1. In Aldor operators can be overloaded, so the definition at step "`%6`" is legal and does not need confirmation. A function is redefined only if a new definition with exactly the same signature is provided.
2. Answering `n` causes the previous definition to be kept.

It is useful to set this option `off` when, for some reason, a file is included a second time[3]. If this file contains some function definitions, you will not then be prompted to confirm each of them.

- `#int timing [on | off]`

[3] Remember to use "`#reinclude`" to include a file that has already been included.

Default is: `on`.

Displays a line after each operation detailing the time taken in the compiler and the interpreter. For example:

```
%28 >> count := 0; for i in 1..1000 repeat count:=count+1
                             Comp: 0 msec, Interp: 10 msec
```

To save space this option has been turned off in the examples given here.

- `#int msg-limit` *number*

Default is: 0 (no limit).

Set the maximum length, in characters, of Aldor messages. This is useful because some of them, such as error messages, could consist of several lines. A value of 0 means that there is no limit. The characters "`...`" at the end of the message will warn you that it has been truncated.

Note that, if you limit the message length, then you will get incomplete messages when you use the interactive mode as a browser (see section 18.3.5). You may also see long type-names cut off with "`...`" in error messages. To see the types in full, use

```
#int options -M no-abbrev
```

- `#int gc`

This command explicitly calls the garbage collector. After the execution, a message showing the amount of memory not released is shown, with additional details if the verbose option is `on`. This operation may take several seconds if the hardware is slow. Note that the garbage collection may occur as needed at other times, even though you have not specifically requested it.

- `#int options` *command-line options*

Set one or more of the options normally available when the compiler is called — for example, optimisations.

Example:

```
#int options -Q3
```

sets the optimisations `on`. Note that, since the interactive mode will be slower for simple expressions if all the optimisations are active but the functions defined are considerably faster, you might want to turn optimisations such as "`-Q3`" `on` before defining time-intensive functions and to turn them `off` or reset them to a lower level (such as "`-Q1`" or "`-Q0`") after the definition.

- `#int cd` *new-directory*

Change the current directory. This is useful if, for example, you want to include files from another directory without typing the path in the "`#include`" directive.

Example: `#int cd /tmp`

- `#int shell "`*shell-command*`"`

Execute the quoted string as a shell command. This is useful, for example, to start an editor session without exiting the interpreter.

Example: `#int shell "vi"`

will start the `vi` editor under Unix.

You can also start a child shell by passing the command which invokes it as a command. Under Unix, for example, you can say:

`#int shell "csh"`

to start a `csh` shell. To return to the interpreter, type `exit`.

18.3 Using the interactive mode

18.3.1 Multi-line input

In order to provide as comfortable an interactive mode as possible, the syntax of the Aldor language, as used interactively, differs slightly from that of the compiled language.

Interactive mode is, by default, indentation sensitive. As a consequence, you do not need a semicolon ("`;`") at the end of each statement — the newline is identified as a separator. This is different from non-interactive use, where you need to use `#pile` to get indentation sensitivity.

Multi-line input can be performed using braces or with a "`==`" as the last symbol on a line.

If you begin a definition with the line

```
...
%4 >> foo(x: Integer): Integer == {
```

then the definition will be considered complete when the number of closing braces ("}") matches the number of opening braces ("{") that you entered. In this case, you must remember that braces always turn off indentation sensitivity so you must use a semicolon ("`;`") to separate your statements.

If you enter

```
...
%4 >> foo(x: Integer): Integer ==
```

Aldor interactive mode is smart enough to understand that you are going to write the body of the function, so the line is not analysed as a complete statement. In this case you need to use spaces or tabs to write your definition with appropriate indentation. The definition will be complete when you start writing at indentation level 0 again. A quick way to terminate the definition of a function when in indentation sensitive mode is to type "`--`" (an empty comment) at the beginning of the line.

We can show some examples. In the first one, a function "`arrToList`" is defined using the indentation sensitive mode. Note that "`==`" is last symbol at the end of line 8.

```
 ...
%5 >> import from Integer, Array Integer, List Integer
%6 >> -- no ';' needed at the end of the line.
%7 >> -- Defining a function using the indentation:
%8 >> arrToList(ar: Array Integer): List Integer ==
        -- the definition is not processed yet and the prompt does
        -- not appear, indicating that you can enter the body of
        -- the function.

        local ls : List Integer := empty   -- no ';'
        for x in ar repeat ls := cons(x,ls)
        ls -- return value

-- This comment at level 0 cause the definition to be processed.
Defined arrToList @ (ar: Array(AldorInteger)) -> List(AldorInteger)
%9 >> -- Here the prompt appears again.
%10 >> a : Array Integer := [1,2,3]
[1,2,3] @ Array(AldorInteger)
%11 >> arrToList a
[3,2,1] @ List(AldorInteger)
```

In the second example the same function is defined using braces to get multi-line input:

```
 ...
%8 >> arrToList(ar: Array Integer): List Integer == {
...        -- Indentation is not necessary, but we suggest using it
...        -- anyway to get a more readable code. Until the definition
...        -- is complete, you need ';' at the end of each line.
...        -- (except when you write comments, of course).
...
...        local ls : List Integer := empty;
...        for x in ar repeat ls := cons(x,ls);
...        ls  -- only here, ';' is not necessary, following
...            -- the language rules.
...
...-- Note that it is essential to match the opening
...-- brace to terminate the function definition -
...-- writing an unindented line does not suffice:
...}
Defined arrToList @ (ar: Array(AldorInteger)) -> List(AldorInteger)
%9 >> -- the prompt appears again.
```

In this case the interpreter begins each newline with three dots to indicate that it is in the middle of a definition. Note that, since the definition is processed only when complete, all the errors will be issued at that point. So, for example, if you forget a terminator in the body of the function in the second example, you will get a syntax error only when you type "}". This is true when using either braces or indentation.

Note: With interactive input it is often convenient to cut and paste to and from a file (for example under X11 and Emacs). If you are going to compile the file and you want to use fragments of code from the interactive mode, you must either insert "`#pile`" at the top of your file or use braces when you (interactively) define multi-line functions.

When you are using the indentation sensitive mode, the interpreter will start to process the code when you add a line with indentation level 0, *unless* this line starts another definition. This makes cutting and pasting of code easier. For example:

```
  ...
%4 >>
arrToList(ar: Array Integer): List Integer ==

        local ls : List Integer := empty   -- no ';'
        for x in ar repeat ls := cons(x,ls)
        ls -- return value


arrToList(ar: Array MachineInteger): List MachineInteger ==

        local ls : List MachineInteger := empty   -- no ';'
        for x in ar repeat ls := cons(x,ls)
        ls -- return value

--
 @
 with
    ==  add ()

%5 >>
```

The previous example also demonstrates that: (1) you can type <Enter> before defining the function (as in the first line), so that the indentation is much nicer; (2) the interactive mode is currently unable to print a list of multiple definitions, so you get a strange message when you close the double definition.

Note: when the construct "`with {...} = add {...}`" is typed on multiple lines, the braces must be placed as in:

```
} == add {
```

and not as in:

```
}
== add {
```

as this will cause a syntax error.

18.3.2 Initialisation file

As explained above, when Aldor starts in interactive mode, it looks in the current directory for a file called `aldorinit.as`. If this file is found, it will be read before the prompt appears. Note that this only happens when Aldor starts in interactive mode ("`-g loop`"), and not for the other modes, such as "`-g run`" and "`-g interp`".

The most common use of the initialisation file is to define macros in order to abbreviate the names of commonly used domains; another use would be to set interactive options (such as `#int history on`) that are `off` by default. An example initialisation file was given earlier.

There is also an alternative method to initialise the interactive environment that, under certain circumstances, is more convenient. You may set the shell environment variable "`INCPATH`" to point to a directory containing an Aldor initialisation file. Suppose that you call this file "`mylib.as`"; then, when you start Aldor with "`-g loop`" you may type:

```
%1 >> #include "mylib"
```

It is usually convenient to add the line:

```
#include "aldor"
```

in the file "`mylib.as`", so that you need to include only this file when you start the interactive mode. Therefore you can initialise the interactive environment by using a file "`aldorinit.as`" in your current directory (that will be loaded automatically when you start) or by using a file in the include path that must be explicitly included when you start (or both).

Two observations:

1. You do not need to type the "`.as`" suffix when you include a file.
2. You do not need to type the directory part of the name when including a file in the include path. Otherwise you do need to specify its directory.

Here is another example of an initialisation file:

```
  #include "aldor"
  #include "aldorinterp"
  MI==>MachineInteger;
  #int verbose off
  #int history on
  out(x) ==> stdout << x;
```

18.3.3 Macros

There are no restrictions for macros. They can be defined and used at top level, as in:

```
%1 >> MI ==> MachineInteger
%2 >> import from MI
```

We recommend defining a set of macros to abbreviate long domain names. These macros, which can be placed, for example, in the initialisation file (see above), can reduce the number of typing errors.

18.3.4 Running inside an editor

If you use editors such as Emacs, we suggest running the interactive mode in an editor inferior shell. This will allow cutting and pasting of definitions into your text and maintaining a history of what was typed, so that in the event of a crash occurring during the interactive mode session you do not lose what you typed.

18.3.5 Using the interactive mode as a browser

The interactive mode also may be used for finding the exports from domains and categories. Suppose that you want to use the `Integer` domain. If you want to know its exports, simply type:

```
%3 >> Integer
```

You will get:

```
 @ Join(
PrimitiveType with
        coerce: BInt -> %
        coerce: % -> BInt
    ==  add ()
,
IntegerType with export to IntegerSegment(%)
    ==  add ()
)
                                           Comp: 20 msec, Interp: 0 msec
```

which is the type of Integer. Now type:

```
%4 >> IntegerType
```

to get:

```
 @ Category == Join(OrderedArithmeticType, BooleanArithmeticType, HashType, InputType, OutputType, SerializableType) with
        bit?: (%, MachineInteger) -> Boolean
        clear: (%, MachineInteger) -> %
        set: (%, MachineInteger) -> %
        coerce: MachineInteger -> %
        machine: % -> MachineInteger
        divide: (%, %) -> (%, %)
        mod: (%, MachineInteger) -> MachineInteger
        mod: (%, %) -> %
        quo: (%, %) -> %
```

```
rem: (%, %) -> %
even?: % -> Boolean
odd?: % -> Boolean
factorial: % -> %
gcd: (%, %) -> %
lcm: (%, %) -> %
integer: Literal -> %
length: % -> MachineInteger
next: % -> %
prev: % -> %
nthRoot: (%, %) -> (Boolean, %)
random: () -> %
random: MachineInteger -> %
shift: (%, MachineInteger) -> %
shift!: (%, MachineInteger) -> %
default
        commutative?: Boolean == true
        lcm(a: %, b: %): % == ..
        next(a: %): % == ..
        prev(a: %): % == ..
        set(a: %, n: MachineInteger): % == ..
        clear(a: %, n: MachineInteger): % == ..
        hash(a: %): MachineInteger == ..
        odd?(a: %): Boolean == ..
        shift!(a: %, n: MachineInteger): % == ..
        ((a: %) mod (n: MachineInteger)): MachineInteger == ..
        even?(a: %): Boolean == ..
        factorial(n: %): % == ..
        ((a: %) mod (b: %)): % == ..
```

and so on.

18.3.6 Loading an Aldor file

If you have an Aldor file, you can load it into the interactive environment, so that, for example, you can interactively call and test defined programs. There are two ways to do this:

1. You can use the `#include "`*myfile*`"` directive. For example, when you start, instead of typing:
   `#include "aldor"`
   you could type:
   `#include "myfile.as"`
   (your file should then contain the `#include "aldor"` directive).
2. If your operating system allows input redirection, you can use this to read a file. For example, in Unix or DOS, you could type:
   `% aldor -g loop < myfile.as`
   (Note that `myfile.as` must respect the braces conventions for the interactive mode, as explained before.)

18.3.7 The # symbol.

In Aldor the "#" symbol is a legal operator: some domains (such as Array) export operations named "#". The problem is that "#" is also used to identify a preprocessor directive, such as "#include" or "#int". This can generate confusion. Suppose, for example, that you type:

```
%1 >> #include "aldor"
%2 >> #include "aldorinterp"
%3 >> import from List MachineInteger, MachineInteger
%4 >> -- create a list of 5 elements
%5 >> l := [1,2,3,4,5]
[1,2,3,4,5] @ List(MachineInteger)
%6 >> #l   -- print the number of elements of 'a'
```

at this point you will get the message:

```
[L5 C1] #1 (Warning) Unknown system command.
```

because the preprocessor is trying to interpret "#a" as a directive. To ensure the correct behaviour, you can simply add a space before "#" when it is not intended as a preprocessor directive, since a line containing a preprocessor directive must start with the "#" (note that the prompt is not considered by the preprocessor). In our case, a space will be the first character of the line. So, if you type:

```
%7 >>  #l  -- with ' ' before '#', you get...
5 @ MachineInteger
```

that is, the correct answer.

18.3.8 Labels

Labels cannot be defined at top level. An input of the following kind is not allowed in interactive mode:

```
%1 >> @lab1
...
%n >> goto @lab1
```

Note: Actually, the interactive environment does not check if the user is trying to jump to a label defined at the top level, so you generally get a segmentation fault if you try to do this. Labels *can* be used within function definitions.

CHAPTER 19

Using Aldor with C

Functions and data structures may be shared between programs written in Aldor and other languages. Here we give simple examples of sharing functions in a mixed Aldor and C programming environment.

Aldor has types corresponding to the primitive C types. These will be described in Section 19.3.

19.1 Using C code from Aldor

For the first example, we show how to call a C function from Aldor. The Aldor file "`arigato.as`" refers to the function "`nputs,`" supplied by the C file "`nputs.c`". Figure 19 shows these files.

The commands to compile these two files and link them together, say on Unix, are:

```
% cc -c nputs.c
% aldor -Fx -laldor arigato.as nputs.o
% ./arigato


Arigato gozaimasu!
Arigato gozaimasu!
Arigato gozaimasu!
```

The first command produces the object file "`nputs.o`". The second command compiles the file "`arigato.as`" and links it with our other file to form an executable program. Finally, the third command runs the resulting program.

The Aldor compiler can make use of C-generated object files, whether they are kept loose or packaged with others in a library archive. The Aldor file using the code or data from C must declare it with the statement "`import ... from Foreign C`": this is the purpose of this statement in "`arigato.as`". When a function is imported from C, a declaration is

```
--
-- arigato.as: A main Axiomxl program calling the C function ‘nputs’.
--
#include "aldor"

MI ==> MachineInteger;

import { nputs: (MI, String) -> MI } from Foreign C;
import from MI, String;

nputs(3, "Arigato gozaimasu!");
```

```
/*
 * nputs.c: A simple C function.
 */
void
nputs(int n, char *s)
{
    int     i;
    for (i = 0; i < n; i++) puts(s);
}
```

Figure 19.1: Aldor code using a C function.

```
/*
 * cside.c:  A main C program calling the A# function ‘lcm’.
 */
#include "foam_c.h"

extern FiSInt   lcm     (FiSInt, FiSInt);

int
main()
{
    printf("The lcm of 6 and 4 is %d\n", lcm(6,4));
    return 0;
}
```

```
--
-- aside.as: An Aldor function made available to C.
--
#include "aldor"

MI ==> MachineInteger;

export { lcm: (MI, MI) -> MI } to Foreign C;

lcm(n: MI, m: MI): MI == (n quo gcd(n,m)) * m;
```

Figure 19.2: C code using an Aldor function.

placed at the head of the generated C file. This declaration is constructed using the data correspondence below.

To call a C function or macro (such as `fputc`) defined in a header file (such as "`<stdio.h>`" or "`myfile.h`") an import of the following form should be used:

```
import { fputc: (MachineInteger, OutFile) -> MachineInteger }
       from Foreign C "<stdio.h>";

import { myfun: MachineInteger -> () }
       from Foreign C "myfile.h";
```

The filename indicates the file to include when generating C. No declaration for `fputc` or `myfun` is produced in the generated C — it is assumed that all the imports are declared (or, in the case of macros, defined).

A "`#include`" line is produced in the generated C for every foreign header file mentioned in the source code, even when no imported function is used. One use of this would be to allow some of the definitions in "`foam_c.h`" to be over-ridden. For example, one could replace the memory management primitives with operations specifically optimised for the current application[1].

19.2 Using Aldor code from C

For the second example, we show how to call an Aldor function from C.

C code which uses Aldor functions should include the file "`foam_c.h`". This file contains the C type definitions which correspond to the various Aldor primitive types. For example, "`FiSInt`" is a `typedef` for the C type corresponding to "`MachineInteger`". On Unix, the full path name for this file is "`$ALDORROOT/include/foam_c.h`".

For this example, the C file "`cside.c`" refers to the function "`lcm,`" supplied by the Aldor file "`aside.as`". These files are shown in figure 19.

The commands to compile, link, and run these files are:

```
% aldor -Fo aside.as
% cc -I$ALDORROOT/include -c cside.c
% cc cside.o aside.o -o lcm64 -L$ALDORROOT/lib -laldor -lfoam -lm
% ./lcm64

The lcm of 6 and 4 is 12
```

The first command compiles the Aldor code in the normal way to produce an object file. On Unix, this produces the object file "`aside.o`".

[1]This is only useful in very unusual circumstances.

Compiling the C code which uses "`aside.o`" requires the use of a "`-I`" option to tell the compiler where to find "`foam_c.h`".

Additional options are needed to link an executable program: a "`-L`" option tells the C compiler where to look for libraries, and the "`-l`" options list the libraries which provide Aldor support functions.

The "`-laldor`" option provides a library with basic Aldor types such as floating point numbers, lists, file I/O, and so on.

The "`-lfoam`" option provides a library with run-time support for such things as memory management and big integer arithmetic. Applications can supply their own run time support library instead, if desired (this involves providing alternative macro definitions to those in "`foam_c.h`" and a C file with whatever code is needed by the macros).

The "`-lm`" option makes the standard C math library available. Because of the way Aldor compiles domains, this generally needs to be included even if no operations from the math library are used.

In "`aside.as`", the line beginning "`export to`" tells the compiler that a wrapper function called `lcm` should be generated for the Aldor function with the same name. This wrapper will convert the C calling convention into that used by Aldor using the rules in the next section. Currently it is possible to export only functions in this way (an Aldor constant can be wrapped in a function, and types have no particular use in C).

19.3 Data correspondence

This section describes the correspondence between the way data values are represented in Aldor and the way they are represented in C. It should be possible from this to understand which Aldor declaration will correspond to a declaration in C, and *vice versa*.

Aldor's abstract machine defines a number of types which correspond to types on the target machine (in this case C on top of some operating system). The "`Machine`" package, described in Section 14.16 on page 148, exports the types provided by the abstract machine. All Aldor values are represented internally as elements of one of these types. The complete listing and definition of the types is given in the FOAM reference guide.

Because many Aldor domains can be parameterized over different types, Aldor uses a pointer-sized object when passing domains. Thus, double precision floating point numbers (which are typically bigger than pointers) are "boxed", and a pointer to the box is passed, rather that the number itself. Types which are the same size or smaller than pointer-size are cast to the pointer type when used in a generic context and cast back as appropriate.

In order to make the underlying types available, Aldor provides the "`Machine`" package, which exports these types and operations on them. For example, the underlying representation type of `DoubleFloat` is

DFlo$Machine[2]. This type should be used when calling foreign functions, and the result coerced back to appropriate generic type at the Aldor level.

Records in Aldor are represented by an aggregate type of some kind in the hosting language. For example, in Scheme a vector is used (and all objects are the same size anyway). In C, structures are used. When calling C-defined functions that use records it is important to ensure that the elements of the Aldor record correspond to elements in the C structure. This implies that records intended for use in Foreign functions should use the underlying types, rather than the user-level types[3].

The Aldor types correspond to C types which are given as `typedef`s in the file "`$ALDORROOT/include/foam_c.h`". The following table shows the correspondance between the types exported from the Aldor package "`Machine`" and C:

Aldor type	C typedef	Usual C type
Nil$Machine	FiNil	Ptr
Word$Machine	FiWord	int
Arb$Machine	FiArb	long int
Ptr$Machine	FiPtr	Ptr
Bool$Machine	FiBool	char
Byte$Machine	FiByte	char
HInt$Machine	FiHInt	short
SInt$Machine	FiSInt	long
Char$Machine	FiChar	char
Arr$Machine	FiArr	Ptr
Rec$Machine	FiRec	Ptr
BInt$Machine	FiBInt	Ptr
SFlo$Machine	FiSFlo	float
DFlo$Machine	FiDFlo	double
A -> *B*	FiClos	struct _FiClos *

Here "`Ptr`" is defined as the type "`char *`" for compatibility with old C dialects, but could equally well be defined as "`void *`".

All other Aldor types defined by the system (*e.g.* in the `libaldor` library) or by a user correspond to the C typedef "`FiWord`". This includes `Integer`, `MachineInteger` and so on. The data correspondence on most 32 bit machines allows one to treat `MachineInteger` and `SInt$Machine` as the same type (which is the reason that "`arigato`", above, works). However, on other machines, for example 16 or 64 bit machines, the two types are not equivalent.

Values belonging to ""Record" types, are pointers to C structs of the corresponding members. For example, the C declaration

[2]or equivalently `BDFlo`, which is a macro defined in "`aldor.as`"

[3]A brief perusal of the file "`$ALDORROOT/samples/lib/libaldorX11`" provides a rather extended example of this.

```
struct {
        int    x;
        short  y;
        double z;
} *r;
```

corresponds to the the Aldor declaration

```
local r: Record(
        x: SInt$Machine,    -- or  x: MachineInteger
        y: HInt$Machine,
        z: DFlo$Machine
);
```

Functions which return no value, or more than one value, are declared to be of type void, and additional return results are returned through pointers passed in as additional arguments.

Thus, the expression:

```
import {
        foo: (Integer, Integer) -> (SInt$Machine, BInt$Machine)
} from Foreign C;
```

implies that foo should be declared (in ANSI C) as:

```
static void foo(FiWord, FiWord, FiSInt *, FiBInt *);
```

A number of examples of exporting and importing C-defined functions can be found in "`$ALDORROOT/samples/test`".

CHAPTER 20

Using Aldor with Fortran-77

20.1 Basics

This section describes how to call subprograms written in Fortran-77 from Aldor, and how to call routines written in Aldor from Fortran-77. Since there is no standard foreign-language interface to Fortran-77 it may be necessary to customise your implementation of Aldor to work with your local Fortran compiler. As is the case with Aldor programs which import other foreign code, programs which use Fortran cannot be run in the interpreter environment.

The current interface supports all of the data types in Fortran-77, and allows Fortran and Aldor functions to be used interchangeably in many contexts. For example a Fortran function can be treated as an Aldor function transparently, and an Aldor routine may behave like a Fortran Function or Subroutine. The only major restriction is that an Aldor program cannot see a Fortran Common block, and a Fortran program has no way of accessing Aldor global variables.

Note that this mechanism uses the C generation facilities described in chapter 19 and does not provide a mechanism for generating native Fortran code. It is principally intended as a means by which an Aldor programmer can make use of the vast body of Fortran code available, and by which a Fortran programmer may embed Aldor code in his or her application.

20.2 Simple Example

This simple example demonstrates the main concepts you need to know to call Fortran from Aldor. It shows the use of a Fortran routine for sorting all or part of a vector of floating-point numbers.

```
1  #include "aldor"
2
3  MINT ==> MachineInteger;
4
5  import {
6    fsort: (Array(DoubleFloat),MINT,MINT,Ref(MINT)) -> ();
7  } from Foreign Fortran;
8
9  import {random : () -> Integer} from RandomNumberSource;
10 import from DoubleFloat;
11
12 -- Set up data
13 error?: MINT := 0;
14 n     : MINT := 10;
15 v     : Array(DoubleFloat) := new(10);
16 for i in 1..n repeat
17   set!(v,i,random()::DoubleFloat/random()::DoubleFloat);
18
19 fsort(v, 1, n, ref error?);
20
21 if zero? error? then {
22   print << "sorted data: " << newline;
23   for i in 1..n repeat print << v.(i::MINT) << newline;
24 }
25
```

line 6–8 Here we import the function from `Foreign(Fortran)`. We will describe the exact correspondence between Aldor and Fortran types later, but for the moment remark that objects whose value will be changed on exit are passed using the `Ref` constructor.

line 20 The Fortran routine is called as if it were an Aldor one. The operator `ref` is used to pass a reference to the error flag (this is described in more detail below). Notice that it is possible to pass numerical data directly (the second argument, which in this case represents the start point of the segment of the vector to be sorted) as well as via a variable.

line 24 Note that the contents of the array have been changed by the Fortran routine.

The program could be compiled as follows, assuming that the `fsort` routine is contained in a library called `sort`.

```
% aldor -Cfortran -Fx -laldor -Clib=sort f77sort.as
```

Note the use of the "`-Cfortran`" flag. This has the effect of causing the linker to link to the appropriate Fortran runtime routines, and may also cause compiler-specific initialisation code to be generated. It is not necessary to use this flag except at the link stage.

20.3 Data Correspondence

Fortran-77 has a fixed and relatively small set of data types, and passes all subprogram parameters by reference (i.e. it passes a pointer to the data rather than a copy of the data). Aldor, on the other hand, has a rich and extensible type system, and in general will pass copies of subprogram data (at least in simple cases). The aim of the interface is to ensure that Foreign functions behave naturally in their host environment.

In practice it would be very restrictive only to be allowed to pass this fixed list of types to Fortran, so Aldor uses a set of categories to indicate that a domain can be used to pass particular types of values. This usually means that the domain's representation is the appropriate basic machine type or a pointer to it. For example the `DoubleFloat` type belongs to the `FortranDouble` category since its representation is a boxed double precision floating point number. Note that none of these categories have any exports, the complete list, and an example of an Aldor type which belongs to each one, is:

Aldor Category	**Fortran Type**	**Example Domain**
`FortranInteger`	INTEGER	`MachineInteger`
`FortranReal`	REAL	`SingleFloat`
`FortranDouble`	DOUBLE PRECISION	`DoubleFloat`
`FortranLogical`	LOGICAL	`Boolean`
`FortranCharacter`	CHARACTER	`Character`
`FortranString`	CHARACTER(*)	`String`
`FortranFString`	CHARACTER(*)	`FixedString`
`FortranComplexReal`	COMPLEX REAL	`Complex(SingleFloat)`
`FortranComplexDouble`	COMPLEX DOUBLE	`Complex(DoubleFloat)`

Fortran strings (or rather Character arrays) are not null-terminated, so to manipulate them it is necessary to know their length. Data from the Aldor domain `String` is automatically converted to the equivalent Fortran object, which is in principle a (length, data) pair. An alternative is to use the `FixedString` type which is parameterised by its length.

By default Aldor passes a copy of a scalar parameter to Fortran, in line with its usual semantics. Since many Fortran routines return results by modifying their arguments there is also a `Ref` constructor which will in effect copy the value of the parameter at the end of the call to the Fortran routine back to the Aldor object (this is often referred to as "copy-in/copy-out" semantics). The choice of whether to declare an argument to a Fortran routine to be e.g. a `DoubleFloat` or a `Ref(DoubleFloat)` is left up to the user, and will depend on whether he or she wishes to inspect its value after the call to Fortran has been completed. Note that it is perfectly safe to declare an argument as e.g. `DoubleFloat` even if it will be modified by Fortran, although of course the modified value will not be visible in Aldor. The `Ref` domain exports two operations to dereference and update instances of itself.

20.3.1 Array Arguments

The normal layout of data in an Aldor array is not suitable for direct use by a Fortran routine because it may contain pointers to its elements, or indeed be a vector of pointers to its rows etc. There is also the important fact that Fortran lays out its arrays in column order whereas languages like Aldor and C tend to use row order.

Supposing that we want to pass an array of objects of domain `T` to Fortran. It simplifies matters greatly if `T` satisfies `DenseStorageCategory`. This means that each element of the array can be stored in a fixed amount of memory and so the array can be constructed as a sequence of objects rather than a sequence of pointers to objects. By default the compiler will try and determine this automatically, but it can be done by hand to improve efficiency.

For an array-like object to be passed to Fortran, it must satisfy either `FortranArray` or `FortranMultiArray` category, depending on whether it is a single-dimensional or multi-dimensional object. These provide exports for converting between the Aldor and Fortran representations, and are *automatically* applied by the compiler. Domains in the standard library which can be used in this way include `PrimitiveArray`, `Array` and `TwoDimensionalArray`.

There are two special cases to these rules. The first concerns arrays of strings, in Aldor we need to ensure that we use a fixed sized string representation, and that the array type satisfies the category `FortranFStringArray`. An example of a domain which can be used in this way is `Array FixedString(10)`.

The second special case arises when we are defining an Aldor function to be passed to (and called from) Fortran. Here we are forced to use `PrimitiveArray` for all array types since it is the Fortran runtime environment rather than Aldor's which will set-up the call. In practice this is not too inconvenient.

20.3.2 Passing Subprograms as Arguments

The use of Fortran `Function` and `Subroutine` arguments is supported. For example:

```
DF ==> DoubleFloat;
import {
        integrate: (DF -> DF, hi: DF, lo: DF) -> DF;
} from Foreign Fortran;

-- integrate the sin function between 0 and 1
integrate(sin, 0.0, 1.0);

-- integrate the cos(fn) function between 0 and 1
foo(fn: DF -> DF): DF == {
        integrand(x: DF): DF == cos fn x;
        integrate(integrand, 0.0, 1.0);
```

```
}
```

In the examples, '`integrate`' is being used with functions which are either exported from a domain or package, or locally defined.

There is one restriction on the way dummy procedures can be used. An Aldor function which calls a Fortran routine cannot be invoked recursively by the call to the Fortran routine.

20.4 Calling Aldor Routines from Fortran

This is very similar to the way Aldor routines can be passed to C functions. There is one restriction — exported functions must be defined in the top level of a file, not within an add-body. Of course the exported function itself may use other functions defined in add bodies so this is not really a problem.

For example, consider the Aldor program:

```
#include "aldor"

import from IntegerPrimesPackage, Integer;

export {ISPRIM:MachineInteger -> Boolean} to Foreign Fortran;

ISPRIM(n:MachineInteger):Boolean == prime?(n::Integer)
```

This creates a function `isprim` which can be called from Fortran. Note that the name of the function must be legal in the Fortran world (strictly speaking this means no more than six characters and uppercase, although most modern compilers take a more relaxed view!). A Fortran routine to call this function might look like:

```
       INTEGER I
       LOGICAL ISPRIM, B
       EXTERNAL ISPRIM
C
       DO 100 I=1,100
         B=ISPRIM(I)
         IF (B) THEN
            PRINT*,I," is prime"
         ENDIF
 100  CONTINUE
C
      END
```

Notice that in Fortran only the return type is given, and that the data type correspondence is the same as that provided earlier.

To compile and link the routines together we might do the following:

```
% aldor -Fo isprime.as
% f77 testprime.f isprime.o -L$ALDORROOT/lib -laldor -lfoam
```

Note that as well as linking to the appropriate Aldor libraries it is important to link to the `foam` library.

20.5 Platform-dependent details

The foreign language interface of Fortran is not standardised, and so there are some details which vary from platform to platform. It is possible to tailor an installation of Aldor to a particular Fortran compiler by editing the file `$ALDORROOT/include/aldor.conf` and setting the values of the following keys:

- `fortran-name-scheme` (indicates how Fortran identifiers are decorated in C: common cases are `underscore` which means an underscore character is added to the end of the name and `no-underscore` which actually means that the identifiers are undecorated);
- `fortran-cmplx-fns` (indicates the protocol for returning Complex values via the name of a function);
- `fortran-libraries` (list of linker options needed to link Fortran programs to Aldor);
- `fortran-io-init-fun` (a function which should be called before any Fortran routines are invoked, typically to initialise streams for standard input and output).

The configuration file is discussed in more detail in chapter 24.

20.6 Larger Examples

This first example shows the use of a Fortran routine to find a root of a function (in this case $e^{-x} - x$) in an interval. Note the use of `Ref` types to return results.

```
#include "aldor"
#include "aldorio"

DF ==> DoubleFloat;
SI ==> MachineInteger;
import
  { root:(DF, DF, DF, DF -> DF, Ref DF, Ref SI} -> () ; }
from Foreign Fortran;

-- Interval containing root
lo : DF := 0.0;
hi : DF := 1.0;

-- Tolerance
eps : DF := 0.00001;

-- Result
x : DF := 0;

errorFlag : SI := -1;

f(x:DF):DF == exp(-x) -x;

root(lo, hi, eps, f, ref x, ref errorFlag);
```

```
if not zero? ifail then stdout << x << newline;
```

The following code calls the NAG library function `e04jyf` to minimise the expression $(x_1 + 10\,x_2)^2 + 5\,(x_3 - x_4)^2 + (x_2 - 2\,x_3)^4 + 10\,(x_1 - x_4)^4$ subject to the constraints:

$$
\begin{array}{rcccr}
1 & < & x_1 & < & 3 \\
-2 & < & x_2 & < & 0 \\
-\infty & < & x_3 & < & \infty \\
1 & < & x_4 & < & 3
\end{array}
$$

It uses the *BasicMath* library to provide the mathematical domains. For more details of what the various arguments do please see the NAG Fortran Library Manual, which is available in both printed and electronic versions.

There are several points to note about this example:

1. When importing the Fortran routine we have named each parameter. This is not strictly necessary but makes the program easier to read.
2. Note that in the import statement we have chosen a particular Aldor type for each parameter. In some cases we are using `Vector` and in others `Array` since this is natural for our application, even although the traget Fortran type is identical in both cases.
3. Note the cunning definition of `objfun`, which makes use of `p` which is defined in an outer scope. Also note the use of `PrimitiveArray`s as parameters (see the note earlier) and the modifiable argument `fc`.
4. The workspace arrays have been constructed using `new`. The category `LinearAggregate` defines an export `empty` which creates an aggregate with no values, but its semantics do not require that any memory be allocated. Thus the use of `empty` should be avoided in cases like this.

```
#include "basicmath"
DF  ==> DoubleFloat;
SI  ==> MachineInteger;
VDF ==> Vector DF;
VSI ==> Vector SI;
ASI ==> Array SI;
ADF ==> Array DF;
POL ==> Polynomial DF;
S   ==> OrderedSymbol;
PDF ==> PrimitiveArray DF;
PSI ==> PrimitiveArray SI;

import {e04jyf : ( n : SI, ibound : SI,
                   funct1 : (SI, PDF, Ref DF, PSI, PDF) -> (),
                   bl : VDF, bu : VDF, x : VDF, f : Ref DF,
                   iw : VSI, liw : SI, w :  VDF, lw : SI,
                   iuser : ASI, user : ADF, ifail : Ref SI) -> () ;
} from Foreign Fortran;

import from DF, SI, VDF, ADF, POL, S, List S, NonNegativeInteger;

minimise(lower:VDF, upper:VDF, start:VDF, p:POL, vars:List S):()=={

  n  : SI  := #lower;

  objfun(nn:SI, xc:PDF, fc:Ref DF, iusr:PSI , usr:PDF):() == {
    vals : List DF := [ xc.i for i in 1..nn ];
    update!(fc, ground eval(p,vars,vals));
  }

  -- Workspace:
  user  : ADF := new(0,0);
  iuser : ASI := new(0,0);
  liw   : SI  := n+2;
  iw    : VSI := new(liw,0);
  lw    : SI  := n*(n-1) quo 2+12@SI*n;
  w     : VDF := new(lw,0);

  -- Output Parameters
  f : DF := 1.0;
  ifail : SI := -1;

  e04jyf(n, 0, objfun, lower, upper, start, ref f, iw, liw, w, lw,
         iuser, user, ref ifail);

  print << "fail = " << ifail << " minimum = " << f << newline;
  print << "minimum point = ";
  for i in 1..n repeat
    outputAsFixed(print, start.i, 8, 4)$FormattedNumericalOutput;

}

lower : VDF := [1.0, -2.0, -1.0e6, 1.0];
upper : VDF := [3.0, 0.0, 1.0e6, 3.0];
start : VDF := [3.0, -1.0, 0.0, 1.0];

-- Set up symbols and build polynomial
vlist : List S := [ +"x1",  +"x2",  +"x3",  +"x4"];
x1 : POL := coerce(+"x1"); x2 : POL := coerce(+"x2");
x3 : POL := coerce(+"x3"); x4 : POL := coerce(+"x4");
p  : POL := (x1+10.0*x2)^(+2)+5.0*(x3-x4)^(+2)+(x2-2.0*x3)^(+4)+
            10.0*(x1-x4)^(+4);
minimise(lower,upper,start,p,vlist);
```

PART IV

Sample Programs

CHAPTER 21

Sample programs

In this chapter we show several examples of Aldor programs. The first few give a brief introduction to the language, followed by some examples using more advanced features of the language. The final few examples show how Aldor can variously emulate or interact with other languages.

This chapter supplements the material in chapter 2 and in part II, the language description. For reference, the examples are as follows:

21.1 Hello

This program prints out a familiar greeting, and then exits. Line 1 allows the use of the base Aldor library in this program. Line 3 prints the greeting. The program shows how a simple program is written, and the syntax for printing objects.

```
1 #include "aldor"
2 #include "aldorio"
3
4 stdout << "Hello, world!" << newline;
```

line 1 Include the file "`aldor.as`". This allows all the domains and categories in the standard library to be used.

line 2 Include the file "`aldorio.as`". This allows to import automatically the necessary types to do some input/output.

line 4 Print the message. "`stdout`" (available because "`aldorio.as`" has been included) is an identifier referring to the console output stream. "`<<`" is an infix operator that prints an object to a given stream, and returns the stream as a value. This allows a cascade of "`<<`" calls.

21.2 Factorial

The next example shows how to define and call functions in Aldor, and a simple form of iteration.

```
1  #include "aldor"
2  #include "aldorio"
3  #pile
4
5  rfact(n: Integer): Integer == if n = 0 then 1 else n*rfact(n-1)
6
7  ifact(n: Integer): Integer ==
8          i := 1
9          while n > 1 repeat
10                 i := i * n
11                 n := n - 1
12         i
```

This program defines two functions for calculating the factorial of an integer. The first shows a simple recursive version, the second is an iterative version.

line 3 Activates the “piling” syntax mode (see Section 4.8 for details).

line 5 Defines a function called `rfact`, which takes an `Integer` and returns the result as an `Integer`

line 7 Defines a function `ifact` with the same signature as `rfact`.

line 8 Assigns the `Integer` `1` to `i`. Note that no declaration is needed if the compiler can infer the type of `i`. In this example, `1` must have been exported from the domain `Integer`. If, for example, `MachineInteger` was also imported, a declaration would have been necessary.

line 9 The `repeat` keyword is used to indicate a loop, and the `while` keyword gives the test for the loop, in this case the loop will exit once `n` is less than or equal to one.

line 10,11 The loop body — multiply `i` and decrement `n` by one. Note that it is legal to assign to a function’s parameters.

line 12 The last statement in the function body gives the value to be returned.

21.3 Greetings

Most operations in Aldor are defined in and exported by *domains*. In order to import from a domain, the `import` statement is used. Imports implicitly happen for the types of parameters of a function, and its return value. The example program below shows the use of the `import` statement, and also how to read user input.

```
1  #include "aldor"
2
3  -- the following imports can be avoided by including "aldorio"
4  import from File;       -- so we can do input
5  import from TextReader; -- for stdin
6  import from TextWriter; -- for stdout
7  import from Character;  -- for newline
8  import from String;     -- for string literals
9
10 -- function to prompt for and return the user's name from the console
11 readName(): String == {
12         stdout << "What is your name?" << newline;
13         line := <<$String stdin;
14         -- delete the trailing newline, and return the result
15         line := [c for c in line | c ~= newline];
16         line;
17 }
18
19 -- main function
20 greet(): () == {
21         name := readName();
22         stdout << "Hello " << name << ", and goodbye..." << newline;
23 }
24
25 greet();
```

This program declares functions to read a name from the console, and then print a personalised note for that name.

line 1 include the standard library

line 4 to 8 import operations from the given domains, allowing us to use operations. Note that the `import from` statement is only needed when operations have not been implicitly imported. In this example, "`aldorio.as`" was not included and so imports for input/output are necessary.

line 11 Defines a function, `readName`. Note that the `()` indicates that the function takes no parameters.

line 20 Defines a function, `greet`. The return type of the function, `()` indicates that the function does not return a value.

line 25 Call the function.

The program produces the following output (user input is in *italic*):

```
What is your name?
Aldor
Hello Aldor, and goodbye...
```

21.4 Cycle

This example demonstrates the manipulation of functions as first-class values, creating new closures over the course of the computation and multiple valued returns.

```
 1 #include "aldor"
 2 #include "aldorio"
 3
 4 import from Integer;
 5
 6 -- Multiple value returns and functional composition.
 7 -- Only creating the closures by * should allocate storage.
 8
 9 I      ==> Integer;
10 III    ==> (I,I,I);
11 MapIII ==> (I,I,I) -> (I,I,I);
12
13 id: MapIII ==
14         (i:I, j:I, k: I): III +-> (i,j,k);
15
16 (f: MapIII) * (g: MapIII): MapIII ==
17         (i:I, j:I, k: I): III +-> f g (i,j,k);
18
19 (f: MapIII) ^ (p: Integer): MapIII == {
20         p < 1  => id;
21         p = 1  => f;
22         odd? p => f*(f*f)^(p quo 2);
23         (f*f)^(p quo 2);
24 }
25
26 -- test routine
27 main(): () == {
28         cycle(a: I, b: I, c: I): III == (c, a, b);
29
30         printIII(a: I, b: I, c: I): () == {
31                 stdout << "a = " << a << " b = "
32                                         << b << " c = " << c << newline
33         }
34         printIII (cycle(1,2,3));
35         printIII (cycle cycle  (1,2,3));
36         printIII ((cycle*cycle)(1,2,3));
37         printIII ((cycle^10)  (1,2,3));
38 }
39
40 main()
41
```

line 9–11 Define some macros as shorthand for some common types used in the program. "`MapIII`" is a map type for functions that take three integers and return three integers.

line 13 The identity function for `MapIII`. Note that "`+->`" produces a closure (otherwise known as a "lambda expression", "anonymous function" or just "function").

line 16 Define a function "`*`" (which is an infix operator) to be a function that given two functions (of type `MapIII`) returns a third function (in this case the composition of the two functions).

line 19–24 Define a function that returns $\underbrace{ff\cdots f}_{n \text{ times}}(a,b,c)$.

The new function is computed by repeated squaring.

line 27–38 A test routine.

line 28, 30 One can define constants inside functions. These have scope local to the function.

The program produces the following output:

```
a = 3 b = 1 c = 2
a = 2 b = 3 c = 1
a = 2 b = 3 c = 1
a = 3 b = 1 c = 2
```

21.5 Generators

Iteration in Aldor is mainly achieved through the use of generators. These are objects representing the state of an iteration, and may be passed around as first-class values. There are two ways of creating generators in Aldor: The `generate` keyword, and the collect form, created by iterators.

```
#include "aldor"

F ==> DoubleFloat;

exp(f: F): F == {
  e: F := 1;
  m: MachineInteger := 1;
  x: F := e;
  for i in 2..12 repeat {
      x := x * f;
      m := m * i;
      e := e +  x/(m::F);
  }
  e;
}

floatSequence(): Generator F == generate {
                x: F := 0.0;
                repeat {
                        yield exp(-x*x);
                        x := x + 0.05;
                }
}

runningMean(g: Generator F): Generator F == {
        n: MachineInteger := 0;
        sum: F   := 0;
        generate {
                for x in g repeat {
                        sum := sum + x;
                        n   := next(n);
                        yield sum/(n::F);
                }
        }
}

step(n: MachineInteger)(a: F, b: F): Generator F == generate {
                m: MachineInteger := prev(n);
                del: F := (b - a)/m::F;
                for i in 1..n repeat {
                        yield a;
                        a := a + del;
                }
}


main(): () == {
        import from F, MachineInteger, TextWriter, Character, String;
        for i in runningMean(x for x in step(11)(0.0, 1.0)) repeat
                stdout << i << newline;

        for i in 1..10 for x in runningMean(floatSequence()) repeat
                 stdout << "next: " << x << newline;
}

main();
```

line 5 Define a helper function which computes an approximated exponential for a `DoubleFloat` value.

line 17 Define a function which creates a generator of `DoubleFloat`s.

line 17 The `generate` keyword introduces the generator body. This is evaluated when a value from the generator is needed, up to the first yield statement, the value of which is returned as the next value for the generator.

line 18–23 The body of the generator. In this case we yield the value of e^{-x^2} for increasing values of x.

line 25 Define a function to calculate the running mean of a sequence.

line 28–34 The generator body. The value is calculated by maintaining a sum of values so far, and dividing by the number of values.

line 37 Define a helper function which returns a generator of `step`s between a and b using n as the size of the interval.

line 49 create a generator and iterate over it. In this case the generator is the running average of $0, 0.1, \ldots, 1.0$.

line 52 where more than one iterator (formed by `for` and `while`) precedes the repeat, iteration is in parallel and terminates when one of the iterators reaches its end condition. In this case we want the first few values of a generator, so the parallel iteration ensures that the loop will complete at or before 10 iterations.

There is a further example on the use of generators on page 220.

21.6 Symbol

Most programming in Aldor is done by defining domains and packages. Here we give a small example of a domain. A package is simply a domain which does not export any operations involving values of type %.

Typically, writing a domain is done in four stages:

1. decide what operations a domain will provide, and from which categories it should inherit,
2. decide how to represent the domain,
3. implement the operations,
4. test the domain.

In this example we show how to define a domain. The example is a symbol datatype which ensures that only a single instance of a given symbol is created.

```
#include "aldor"
#include "aldorio"

define BasicType: Category == Join(OutputType, PrimitiveType);

Symbol: BasicType with {
        name:   % -> String;    ++ the name of the symbol
        coerce: String -> %;    ++ conversion operations
        coerce: % -> String;
} == add {
        Rep    == String;

        import from Rep, Pointer;

        local symTab: HashTable(String, %) := table();

        name(sym: %): String == sym::String;

        coerce(sym: %): String == rep(sym);

        coerce(s: String): % == {
                symb?: Partial(%) := find(s, symTab);
                import from Boolean;
                not failed? symb? => retract symb?;
                str := copy s;
                set!(symTab,str,per str);
                per str;
        }

        (s1: %) = (s2: %): Boolean == rep s1 = rep s2;
        (p: TextWriter) << (sym: %): TextWriter == {
                        p << "'" << sym::String << "'";
        }
}

Test(): () == {
        import from Symbol;

        stdout << "hello"::Symbol << newline;
}
Test()
```

line 4 Define the category `BasicType` which is the union of PrimitiveType and OutputType.

line 6 Define symbol to be a constant with the given type — in this case `Symbol` is a `Domain` which implements `BasicType` and provides 3 additional operations.

line 6–9 Signatures for operations on the domain. `%` in the type refers to “this `Domain`”.

line 10 The `add` expression creates a domain from a sequence of definitions.

line 11 Define how `Symbols`, our new type are represented. In this case, we just use a string. If the signature required that additional information was stored on the symbol we might want to use a different representation.

line 13 Allow operations from the representation domain, *i.e.* `String` to be used.

line 15 `%` can be used inside the add body to refer to the current domain.

line 15 define a local variable for storing the symbol table.

line 17–34 Define functions exported by the domain.

line 19 and 26 The operation `rep` allows a value of type % to be treated as if it were of type `Rep`. `per` is the inverse operation, *i.e.* it converts an object of type `Rep` into type %. These two operations are currently macros[1]. See Section 2.4.

[1] `per` can be thought of as ‘rep backwards’, or as ‘*per*cent’.

21.7 Stack

It is possible to define a function whose return type is a domain. In this case, the result is called a *parameterised domain*. The example is a simple stack with a few operations for creating new stacks, as well as pushing and popping values from an existing stack.

```
#include "aldor"
#include "aldorio"

-- implementation of stacks via lists
-- the lines starting with ++ are saved in the output of
-- the compiler, and may be browsed with an appropriate tool

Stack(S: OutputType): OutputType with {
        empty?:    % -> Boolean; ++ test for an empty stack
        empty:    () -> %;       ++ create an empty stack
        push!:(S, %) -> %;       ++ put a new element onto the stack
        pop!:      % -> S;       ++ remove the top element and return it
        top:       % -> S;       ++ return the top of the stack

        export from S;
                -- expose all operations from S
                -- when Stack S is imported
} == add {
        -- Stacks are represented using a list.
        -- To go between the representation and % we use the
        -- rep and per functions.
        Rep == Record(contents: List S);
        import from Rep;

        -- utility functions
        local contents(stack: %): List S == rep(stack).contents;

        -- simple functions
        empty(): % == per [empty];
        empty?(s: %): Boolean == empty? contents s;
        top(s: %): S == first contents s;

        push!(elt: S, s: %): % == {
                rep(s).contents := cons(elt, contents s);
                s
        }

        pop!(s: %): S == {
                next := first contents s;
                rep(s).contents := rest contents s;
                next;
        }

        -- needed to satisfy OutputType
        import from String;
        (tw: TextWriter) << (s: %): TextWriter == tw << "<stack>";
}


test(): () == {
        -- Importing the domains involed in the next two
        -- lines is made by the affectations.
        l: List MachineInteger := [1,2,3,4,5,6];
        stack: Stack MachineInteger := empty();
        for x in l repeat
                push!(x, stack);
```

```
57          -- Importing the domains involed in the next
58          -- line is needed.
59          stdout << "stack is:" << stack << newline;
60          while not empty? stack repeat {
61                  stdout << "Next is: " << top stack << newline;
62                  pop! stack;
63          }
64  }
65
66  test()
67
```

The chosen representation type is a list over the same domain as the stack. This allows us to implement the operations with minimum complications. A better representation might be a linked list of arrays, but this would clutter the example more than necessary.

line 8 Define `Stack` to be a function that takes a BasicType (*i.e.* most of the domains in Aldor), and returns an object which satisfies BasicType, and additionally provides some stack operations. The interface is specified by the `with` construct.

line 9 Declare an operation ‘empty?’ which takes a value from the current domain as an argument, and returns a Boolean value. The line starting ‘++’ is a description comment, and is saved along with the declaration of `empty?` in the intermediate file.

line 16 The ‘export from’ statement indicates that all operations exported by `S` should be imported when Stream S is imported.

line 23 Define the representation of the Stack. This form of a define statement (without a declaration) implies that the type of Rep is exactly that of the type of the right hand side of the expression, *i.e.* `Record(contents: List S)`.

line 24 Allow operations from Rep to be used. We do not need to say `import from List S`, as Record exports its arguments.

line 27 Define a function which returns the internal list of values maintained by the stack.

line 30–43 Define the operations required explicitly by the category.

line 46–47 Define the operations needed to satisfy inherited operations.

Once the domain is defined, it may be tested. Aldor's interactive loop, "`aldor -G loop`" is useful here.

```
% aldor -G loop
%1 >> #include "stack.as"
stack is:<stack>
Next is: 6
Next is: 5
Next is: 4
Next is: 3
Next is: 2
Next is: 1
%2 >> import from Stack MachineInteger
%3 >> s: Stack MachineInteger := empty()
<stack> @ Stack(MachineInteger)
%4 >> push!(12, s)
<stack> @ Stack(MachineInteger)
%5 >> top s
12 @ MachineInteger
%5 >> pop! s
12 @ MachineInteger
```

We should probably test it a little further (*e.g.* boundary conditions), but this gives the general idea.

21.8 Recursive structures

This program shows a recursively defined data type: the type of binary trees parameterised with respect to the type of data placed on the interior nodes. This tree type provides several generators which allow the trees to be traversed in different ways.

```
#include "aldor"
#include "aldorio"

Tree(S: OutputType): OutputType with {
    export from S;

    empty: %;
    tree:  S  -> %;
    tree:  (S, %, %) -> %;

    empty?: % -> Boolean;

    left:   % -> %;
    right:  % -> %;
    node:   % -> S;

    preorder:  % -> Generator S;
    inorder:   % -> Generator S;
    postorder: % -> Generator S;
}
== add {
    Rep == Record(node: S, left: %, right: %);
    import from Rep;

    empty: % == nil$Pointer pretend %;
    empty?(t: %): Boolean == nil?(t pretend Pointer)$Pointer;

    tree(s: S): % == per [s, empty, empty];
    tree(s: S, l: %, r: %): % == per [s, l, r];

    local nonempty(t: %): Rep == {
        import from String;
        empty? t => error "Taking a part of a non-empty tree";
        rep t
    }

    left (t: %): % == nonempty(t).left;
    right(t: %): % == nonempty(t).right;
    node (t: %): S == nonempty(t).node;

    preorder(t: %): Generator S == generate {
        if not empty? t then {
            yield node t;
            for n in preorder left  t repeat yield n;
            for n in preorder right t repeat yield n;
        }
    }
    inorder(t: %): Generator S == generate {
        if not empty? t then {
            for n in inorder left  t repeat yield n;
            yield node t;
            for n in inorder right t repeat yield n;
        }
    }
    postorder(t: %): Generator S == generate {
        if not empty? t then {
            for n in postorder left  t repeat yield n;
```

```
            for n in postorder right t repeat yield n;
            yield node t;
        }
    }
    (tw: TextWriter) << (t: %): TextWriter == {
        import from String;
        import from S;

        empty? t => tw << "empty";
        empty? left t and empty? right t => tw << "tree " << node t;

        tw << "tree(" << node t << ", "
           << left t  << ", " << right t << ")"
    }
}


main():() == {
    import from Tree String;
    import from List String;

    t := tree("*", tree("1", tree "a", tree "b"),
           tree("2", tree "c", tree "d"));

    stdout << "The tree is " << t << newline;
    stdout << "Preorder:   " << [preorder  t] << newline;
    stdout << "Inorder:    " << [inorder   t] << newline;
    stdout << "Postorder:  " << [postorder t] << newline;
}

main();
```

When compiled and run, this program gives the following output:

```
The tree is tree(*, tree(1, tree a, tree b), tree(2, tree c, tree d))
Preorder:   [*,1,a,b,2,c,d]
Inorder:    [a,1,b,*,c,2,d]
Postorder:  [a,b,1,c,d,2,*]
```

21.9 Swap

Higher order functions which construct types are first-class values. This example shows how to swap structure layers in a data type by using higher order functions as parameters to a generic program.

```
1  #include "aldor"
2  #include "aldorio"
3  #pile
4
5  I  ==> MachineInteger;
6  Ag ==> (S: Type) -> BoundedFiniteLinearStructureType S;
7
8  -- This function takes two type constructors as arguments and
9  -- produces a new function to swap aggregate data structure layers.
10
11 swap(X:Ag,Y:Ag)(S:Type)(x:X Y S):Y X S ==
12     import from Y S, X S
13     [[s for s in y]for y in x]
14
15 import from I, List(I);
16
17 -- Form an array of lists:
18 al: Array List I := [[i+j-1 for i in 1..3] for j in 1..3]
19
20 stdout << "This is an array of lists: " << newline
21 stdout << al << newline << newline
22
23 -- Swap the structure layers:
24
25 la: List Array I := swap(Array,List)(I)(al)
26
27 stdout << "This is a list of arrays:  " << newline
28 stdout << la << newline
```

line 6 Define a macro "`Ag`" as a shorthand for the type which takes a Type as an argument, and returns a second type which is a `BoundedFiniteLinearStructureType` over it.

line 11–13 Define the "`swap`" function. This curried definition exchanges the "`X`" and "`Y`" layers in a structure. This function is written generically to use

- "`generator`" from `X Y S` for the outer iteration,
- "`generator`" from `Y S` for the inner iteration,
- "`bracket`" from `X S` as the inner constructor,
- "`bracket`" from `Y X S` as the outer constructor.

line 25 Call "`swap`" to exchange the `Array` and `List` layers.

When executed via "`aldor -G run swap.as`," the following output is produced.

```
This is an array of lists:
[[1,2,3],[2,3,4],[3,4,5]]

This is a list of arrays:
[[1,2,3],[2,3,4],[3,4,5]]
```

21.10 Objects

In Aldor, values are *not* self-identifying — there is no way of retrieving a given value's type from the value itself.

We can implement this functionality in the "`Object`" datatype, which holds both a value and its type.

The following example shows the implementation of Object and a use of it.

```
1  #include "aldor"
2  #include "aldorio"
3
4  -- OutputType objects ------------------------------------------------
5  --
6  -- These objects can be printed because each belongs to some OutputType.
7  --
8
9  Object(C: Category): with {
10         object:         (T: C, T) -> %;
11         avail:          % -> (T: C, T);
12 }
13 == add {
14         Rep == Record(T: C, val: T);
15         import from Rep;
16
17         object  (T: C, t: T) : %        == per [T, t];
18         avail   (ob: %) : (T: C, T)     == explode rep ob;
19 }
20
21 main():() == {
22     import from Integer, List Integer;
23     bobfun(bob: Object OutputType): () == {
24         f avail bob where
25         f(T: OutputType, t: T) : () == {
26                 stdout << "This prints itself as: " << t << newline;
27         }
28     }
29     import from Object OutputType;
30     boblist: List Object OutputType := [
31         object (String,       "Ahem!"),
32         object (Integer,      42),
33         object (List Integer, [1,2,3,4])
34     ];
35     for bob in boblist repeat bobfun bob;
36 }
37
38 main();
39
```

line 9 Define the `Object` datatype. As we can see there is a constructor called `object` to bundle a value and its type and a "dismantler" called `avail` to get the value and the type from a previously constructed object. This domain is parameterized by a category called `C`. All types that we can bundle in an `Object(C)` will satisfy the category `C`.

line 23 Define a function, "`bobfun`", to take object arguments. The arguments are objects whose underlying type satisfies the category `OutputType`.

line 24 Use the "`avail`" operation to split an object into its type and value components, then call the local function "`f`" on the dependent type/value pair.

line 26 Print the object value. The "`<<`" operation is available, because each object's type satisfies `OutputType`.

line 30 Form a list of `OutputType` objects. Each is formed with the "`object`" function from `Object(OutputType)`.

line 35 Call "`bobfun`" on each of the objects in the list.

When run with "`aldor -G run objectb.as`" this program produces the following output:

```
This prints itself as: Ahem!
This prints itself as: 42
This prints itself as: [1,2,3,4]
```

The richer the category argument to `Object`, the more interesting operations may be performed on the object values. A second example of using `Object` is shown below. In this case each object value belongs to some ring, and this fact is used in the arithmetic calculation.

```
#include "aldor"
#include "aldorio"

-- Arithmetic objects -----------------------------------------------------
--
-- The objects have arithmetic because each belongs to ArithmeticType.
--

Object(C: Category): with {
        object:         (T: C, T) -> %;
        avail:          % -> (T: C, T);
}
== add {
        Rep == Record(T: C, val: T);
        import from Rep;

        object  (T: C, t: T) : %        == per [T, t];
        avail   (ob: %) : (T: C, T)     == explode rep ob;
}


main():() == {
    import from MachineInteger, Integer;
    robfun(rob: Object IntegerType): () == f avail rob  where {
        f(T: IntegerType, r: T): () == {

            -- Object-specific arithmetic:
            s := (r + 1)^3;
            t := (r - 1)^4;
            u := s * t;

```

```
32             -- Object-specific output:
33             stdout << "r = " << r << newline;
34             stdout << "    s = (r + 1) ^ 3 = " << s << newline;
35             stdout << "    t = (r - 1) ^ 2 = " << t << newline;
36             stdout << "    s * t = "         << u << newline;
37
38             -- Can check for additional properties and use if there.
39             if T has TotallyOrderedType  then {
40                 stdout << "The result is ";
41                 if u < 0 then stdout << "negative";
42                 if u > 0 then stdout << "positive";
43                 if u = 0 then stdout << "zero";
44                 stdout << newline;
45             }
46             else
47                 stdout << "No order for this object." << newline;
48
49             stdout << newline;
50         }
51     }
52     import from DoubleFloat, Integer;
53     import from Object IntegerType;
54     roblist: List Object IntegerType := [
55         object (     Integer, -42),
56         object (      MachineInteger, -42)
57     ];
58     for rob in roblist repeat robfun rob
59 }
60
61 main();
```

line 24 Define a function, "`robfun,`" to take object arguments. This time the arguments are objects whose type slots satisfy the category `IntegerType`. Again "`avail`" is used to split an object into its component parts (type and value).

line 28–30 Perform various arithmetic operations on the value. All of "`+`" "`-`" "`*`" "`^`" and "`1`" are provided by the particular object.

line 33–36 Print the results of the arithmetic. This is possible because each `IntegerType` provides a "`<<`" operation.

line 39 The `has` operator tests whether a given domain satisfies a particular category. This test is made at run-time.

line 40–44 Inside an `if` statement, if it can be deduced that an imported domain satisfies an additional category (using the information in the evaluation of the `if` expression), then the additional operations are made available within the "`then`" branch of the `if` statement. In this case, "`<`" and "`>`" are available because `T` is seen to also satisfy `TotallyOrderedType`.

The output produced when running this program with the command "`aldor -G run objectb.as`" is shown below.

```
r = -42
    s = (r + 1) ^ 3 = -68921
    t = (r - 1) ^ 2 = 3418801
    s * t = -235627183721
```

```
The result is negative

r = -42
    s = (r + 1) ^ 3 = -68921
    t = (r - 1) ^ 2 = 3418801
    s * t = 596017559
The result is positive
```

21.11 Mandel

The next example shows the use of machine-level floating point in Aldor. This program would be a bit simpler if we first implemented a Complex domain.

```
#include "aldor"
#include "aldorio"

MI ==> MachineInteger;
F  ==> DoubleFloat;

step(n: MachineInteger)(a: F, b: F): Generator F == generate {
    m: MachineInteger := prev(n);
    del: F := (b - a)/m::F;
    for i in 1..n repeat {
        yield a;
        a := a + del;
    }
}

default minR, maxR, minI, maxI: F;
default numR, numI, maxIters:   MI;
default drawPt: (r: MI, i: MI, n: MI) -> ();

drawMand(minR, maxR, numR, minI, maxI, numI, drawPt, maxIters): () == {

    mandel(cr: F, ci: F): MI == {
       zr: F := 0;
       zi: F := 0;
       n: MI := 0;
       while (zr*zr + zi*zi) < 4.0 for free n in 1..maxIters repeat {
          zr := zr*zr -zi*zi + cr;
          zi := 2.0*zi*zr + ci;
       }
       return n;
    }

    for i in step(numI)(minI, maxI) for ic in 1..numI repeat
      for r in step(numR)(minR, maxR) for rc in 1..numR repeat
        drawPt(rc, ic, mandel(r,i));
}

import from F;
maxN: MI == 100;
maxX: MI == 25;
maxY: MI == 25;

drawPoint(x: MI, y: MI, n: MI): () =={
  if      n = maxN then stdout << "   ";
  else if n < 10   then stdout << "  " << n;
  else                  stdout << " "  << n;
  if x = maxX then stdout << newline;
}

drawMand(-2.0, -1.0, maxX, -0.5, 0.5, maxY, drawPoint, maxN);

```

Machine level operations are done inline when the optimiser is used while compiling (use the options "`-Q3 -Qinline-limit=10`"). This has the result that the generated code speed is comparable with that of the equivalent code in languages such as C.

21.12 Integers mod n

This example shows how add-inheritance can be used in the implementation of integers modulo a particular number. We first define a category and a generic member of the category, called `ModularIntegerNumberRep`. We then have two specific instances of the category which inherit most of their operations from the generic domain. Note that in the definition of `MachineIntegerMod` we over-ride the generic multiplication with something more efficient. Also note that the definition of `Rep` is required in both the specific implementations so that the `rep` and `per` macros will work, however it is essential that it is compatible with the `Rep` in the generic case.

```
-----------------------------------------------------------------
----
---- imod.as: Modular integer arithmetic.
----
-----------------------------------------------------------------

#include "aldor"

define ModularIntegerType(I: IntegerType):Category ==
  ArithmeticType  with {
    integer:Literal -> %;
    coerce: I -> %;
    lift:   % -> I;
    inv:    % -> %;
    /:      (%, %) -> %;
}


ModularIntegerNumberRep(I: IntegerType)(n: I):
    ModularIntegerType(I) with
== add {
    Rep  == I;
    import from Rep;

    0: % == per 0;
    1: % == per 1;

    (^)(x:%,n:MachineInteger):% ==
        binaryExponentiation!(x, n)$BinaryPowering(%,MachineInteger);

    (x: %) * (y: %): % == per((rep x * rep y) mod n);
    commutative?: Boolean == true;

    (port: TextWriter) << (x: %): TextWriter == port << rep x;

    coerce (i: I): %          == per(i mod n);
    integer(l: Literal): %    == per(integer l mod n);
    lift   (x: %): I          == rep x;

    zero?(x: %): Boolean      == x = 0;
    (x: %) = (y: %): Boolean  == rep(x) = rep(y);
    (x: %)~= (y: %): Boolean  == rep(x) ~= rep(y);

    - (x: %): %        == if x = 0 then 0 else per(n - rep x);
    (x: %) + (y: %): % ==
        per(if (z := rep x-n+rep y) < 0 then z+n else z);
    (x: %) - (y: %): % ==
        per(if (z := rep x  -rep y) < 0 then z+n else z);
```

```

    (x: %) / (y: %): % == x * inv y;

    inv(j: %): % == {
        local c0, d0, c1, d1: Rep;
        (c0, d0) := (rep j, n);
        (c1, d1) := (rep 1, 0);
        while not zero? d0 repeat {
            q := c0 quo d0;
            (c0, d0) := (d0, c0 - q*d0);
            (c1, d1) := (d1, c1 - q*d1)
        }
        assert(c0 = 1);
        if c1 < 0  then c1 := c1 + n;
        per c1
    }
}


SI ==> MachineInteger;
Z ==> Integer;

MachineIntegerMod(n: SI): ModularIntegerType(SI) with {
    lift: % -> Z;
} == ModularIntegerNumberRep(SI)(n) add {
    Rep == SI;

    lift(x: %): Z == (rep x)::Z;

    (x: %) * (y: %): % == {
        import from Machine;
        (xx, yy) := (rep x, rep y);
        xx = 1 => y;
        yy = 1 => x;
        -- Small case
        HalfWord ==> 32767; --!! Should be based on max$Rep
        (n < HalfWord) or (xx<HalfWord and yy<HalfWord) => (xx*yy)::%;

        -- Large case
        (nh, nl) := double_*(xx pretend Word, yy pretend Word);
        (qh, ql, rm) := doubleDivide(nh, nl, n pretend Word);
        rm pretend %;
    }
}


IntegerMod(n: Z): ModularIntegerType(Z) with {
    coerce: SI -> %;
} == ModularIntegerNumberRep(Z)(n) add {

    coerce(i:SI):% == (i::Z)::%;
}

```

21.13 Extensions

Aldor allows the library types to be extended with new operations. For example, one may wish to add a `DifferentialRing` category to the language. The extension mechanism allows existing domains, such as `Integer` and Polynomial to include these new categories. The extension mechanism operates via the "`extend`" keyword.

The following example allows us to sort lists of symbols. The `List(S)` domain exports a `sort` operator if `S` belongs to the category `TotallyOrderedType`. Although `Symbol` does not belong to this category `String` does, and we can use this fact to implement the necessary exports in a fairly straightforward manner.

```
1  #include "aldor"
2  #include "aldorio"
3
4  MI ==> MachineInteger;
5
6  extend String:TotallyOrderedType with { } == add {
7          import from Character, MI;
8
9          (<)(u:%, v:%):Boolean == {
10              (a: MI, b: MI) := (#u, #v);
11              zero? a => not zero? b;
12              zero? b => false;
13              for i in 0..min(a,b) repeat {
14                  u.i < v.i => return true;
15                  u.i > v.i => return false;
16              }
17              a < b;
18          }
19         (>)(u:%, v:%):Boolean == v < u;
20         (<=)(u:%, v:%):Boolean == not (u > v);
21         (>=)(u:%, v:%):Boolean == not (v > u);
22         min(u:%, v:%):% == if u < v then u else v;
23         max(u:%, v:%):% == if u < v then v else u;
24 }
25
26 import from List String;
27
28 l1 := ["animal","aldor","apple","anaconda","atlantic"];
29 l2 := sort! copy l1;
30
31 stdout << l1 << newline;
32 stdout << l2 << newline;
```

line 6–24 Extend Symbol. In order to satisfy `TotallyOrderedType` we need to provide six operations which in this case we implement in terms of strings.

line 26–32 Test the extended domain constructor.

21.14 Text input

In the next example, the Aldor `TextReader` and `TextWriter` datatypes are used to provide a number of useful text processing operations.

```
#include "aldor"

SI   ==> MachineInteger;
Char ==> Character;

-- Asserts that the charecter is a vowel
vowel?(c: Char): Boolean ==  {
    import from String;
    c = char "a" or c = char "e" or c = char "i"
    or c = char "o" or c = char "u";
}

-- Remove all the vowels from the input and
-- write the result on the output
removeVowels(tr: TextReader, tw: TextWriter): () == {
    import from Char, TextWriter, String;
    c: Char := read! tr;
    while (c ~= eof) repeat {
      if not vowel? c then write!(c, tw);
      c := read! tr;
    }
}

-- Prints a string and a newline on the standard output
printMessage(s:String):() == {
     import from String, TextWriter, Char;
     stdout << s << newline;
}

-- Constructs a string from the characters in the list
-- given its length and assuming that this list needs
-- to be reversed
convert(l: List Char, n: SI): String == {
      import from SI, Char;
      s: String := new(n, space);
      while (not empty? l) for i in 1..n repeat {
          c := first l; l := rest l;
          s.(n-i) := c;
      }
      s;
}

-- Generates the lines (as strings) from the input
lines(tr:TextReader): Generator String == generate {
     import from String;
     printMessage("entering lines");
     c: Char := read! tr; l: List Char := []; n: SI := 0;
     while (c ~= eof) repeat {
         l := []; n:=0;
         while (c ~= newline) repeat {
             l := cons(c,l); n := n + 1; c := read! tr;
         }
         yield convert(l,n); c := read! tr;
     }
     printMessage("leaving lines");
}

main(): () == {
    import from String;
    printMessage("entering test");
```

```
61      f1: File := open("/etc/passwd",fileRead);
62      f2: File := open("/tmp/passwd",fileWrite);
63      tr: TextReader := f1::TextReader;
64      tw: TextWriter := f2::TextWriter;
65      removeVowels(tr,tw);
66      close! f1;
67      close! f2;
68      f1: File := open("/tmp/passwd",fileRead);
69      tr := f1::TextReader;
70      for l in lines(tr) repeat {
71          printMessage(l);
72      }
73      close! f1;
74      printMessage("leaving test");
75  }
76
77  main();
78
```

line 7 Define a predicate for testing whether a given character is a vowel

line 15 Define a function to print the non-vowel characters in a file.

line 15 `TextReader` provides a generator on a reader which returns the sequence of characters in the reader. `TextWriter` provides primitives to write on a stream.

line 19 `write!` puts a single character onto a stream. The stream is passed as a parameter with the name `tw`. A possible value for `tw` would be `stdout` which is a value of type `TextWriter` and thus an output stream. It is attached to the default output device of the process.

line 25 Define a function to print a text message.

line 33 Define a conversion function from a list of `Character`s to a String.

line 44 `lines` creates a generator which returns the contents of an input stream (passed as a `TextReader`) as a sequence of lines terminated by `newline`.

line 61 Open a file for input. The value `fileRead` is exported by `File` and is the mode chosen to open the file.

line 62 `fileWrite` is also exported by `File`.

line 63 The file `f1` is `coerce`d to an input stream of type `TextReader`.

line 64 The file `f2` is `coerce`d to an output stream of type `TextWriter`.

21.15 Quanc8

The next example gives a Fortran-style program for numeric integration. The program demonstrates how an algorithm described in the pre-structured programming era may be transcribed without introducing errors by reworking its logic. The program was transcribed from the textbook described in the first comment, and produced correct values on its first run. Of course, if you have access to a callable library containing the routines it should be possible to import the operations directly into Aldor.

The "`goto`" construct in Aldor takes the name of a label, and transfers control to that label. Labels are introduced by the "`@`" symbol.

```
#include "aldor"

R ==> DoubleFloat;
I ==> MachineInteger;

+++ quanc8: Quadrature, Newton-Cotes 8-panel
+++
+++ (This is a literal translation of the Fortran program given
+++  in ``Computer Methods for Mathematical Computations'' by Forsythe,
+++  Malcolm and Moler, Prentice-Hall 1977.)
+++
+++ Estimate the integral of fun(x) from a to b to a given tolerance.
+++ An automatic adaptive routine based on the 8-panel Newton-Cotes
+++ rule.
+++
+++ Input:
+++   fun     The name of the integrand function subprogram f(x).
+++   a       The lower limit of integration.
+++   b       The upper limit of integration. (b may be less than a.)
+++   relerr  A relative error tolerance. (Should be non-negative)
+++   abserr  An absolute error tolerance. (Should be non-negative)
+++
+++ Output:
+++   result  An approximation to the integral hopefully satisfying
+++           the least stringent of the two error tolerances.
+++   errest  An estimate of the magnitute of the actual error.
+++   nofun   The number of function values used in the calculation of
+++           the result.
+++   flag    A reliability indicator.  If flag is zero, then result
+++           probably satisfies the error tolerance.  If flag is
+++           xxx.yyy then xxx = the number of intervals which have
+++           not converged and 0.yyy = the fraction of the interval
+++           left to do when the limit on nofun was approached.

quanc8(fun: R -> R, a: R, b: R, abserr: R, relerr: R):
      (Xresult: R, Xerrest: R, Xnofun: I, Xflag: R)
== {
        local result, errest, flag: R;
        local nofun: I;
        RETURN ==> return (result, errest, nofun, flag);

        local w0, w1, w2, w3, w4, area, x0, f0, stone, step, cor11: R;
        local qprev, qnow, qdiff, qleft, esterr, tolerr, temp: R;
        default i, j : I;

        qright: Array R       := new(31, 0.0);
        f:      Array R       := new(16, 0.0);
        x:      Array R       := new(16, 0.0);
```

```
        fsave:  Array Array R := [new(30, 0.0) for i in 1..8];
        xsave:  Array Array R := [new(30, 0.0) for i in 1..8];

        local levmin, levmax, levout, nomax, nofin, lev, nim: I;

        --
        -- *** Stage 1 ***    General initializations
        -- Set constants
        --
        levmin := 1;
        levmax := 30;
        levout := 6;
        nomax  := 5000;
        nofin  := nomax - 8 * (levmax - levout + 2 ^ (levout + 1));
        --
        -- Trouble when nofun reaches nofin
        --
        w0 :=   3956.0 / 14175.0;
        w1 :=  23552.0 / 14175.0;
        w2 :=  -3712.0 / 14175.0;
        w3 :=  41984.0 / 14175.0;
        w4 := -18160.0 / 14175.0;
        --
        -- Initialize running sums to zero.
        --
        flag := 0.0;
        result := 0.0;
        cor11 := 0.0;
        errest := 0.0;
        area := 0.0;
        nofun := 0;
        if a = b then RETURN;
        --
        -- *** Stage 2 ***   Iniitalization for first interval
        --
        lev := 0;
        nim := 1;
        x0  := a;
        x(16) := b;
        qprev := 0.0;
        f0 := fun(x0);
        stone := (b - a)/16.0;
        x(8)  := (x0   + x(16)) / 2.0;
        x(4)  := (x0   + x(8) ) / 2.0;
        x(12) := (x(8) + x(16)) / 2.0;
        x(2)  := (x0   + x(4) ) / 2.0;
        x(6)  := (x(4) + x(8) ) / 2.0;
        x(10) := (x(8) + x(12)) / 2.0;
        x(14) := (x(12)+ x(16)) / 2.0;
        for j in 2..16 by 2 repeat
                f(j) := fun(x(j));
        nofun := 9;
        --
        -- *** Stage 3 ***   Central calculation
        -- Requires qprev,x0,x1,...,x16,f0,f2,f4,...,f16.
        -- Calculates x1,x3,...x15, f1,f3,...f15,qleft,qright,
        --            qnow,qdiff,area.
        --
@L30    x(1) := (x0 + x(2)) / 2.0;
        f(1) := fun(x(1));
        for j in 3..15 by 2 repeat {
                x(j) := (x(j-1) + x(j+1)) / 2.0;
                f(j) := fun(x(j));
        }
```

```
        nofun := nofun + 8;
        step := (x(16) - x0) / 16.0;
        qleft := (w0*(f0+f(8)) + w1*(f(1)+f(7)) + w2*(f(2)+f(6)) +
                  w3*(f(3)+f(5)) + w4*f(4)) * step;
        qright(lev+1) := (w0*(f(8) + f(16))+w1*(f(9)+f(15))+w2*(f(10)+
                  f(14)) + w3*(f(11)+f(13)) + w4*f(12)) * step;
        qnow := qleft + qright(lev+1);
        qdiff := qnow - qprev;
        area := area + qdiff;
        --
        -- *** Stage 4 ***   Interval convergence test
        --
        esterr := abs(qdiff) / 1023.0;
        tolerr := max(abserr, relerr*abs(area)) * (step/stone);
        if lev < levmin then goto L50;
        if lev >=levmax then goto L62;
        if nofun > nofin then goto L60;
        if esterr <= tolerr then goto L70;
        --
        -- *** Stage 5 ***   No convergence
        -- Locate next interval
        --
@L50    nim := 2*nim;
        lev := lev + 1;
        --
        -- Store right hand elements for future use.
        --
        for i in 1..8 repeat {
                fsave(i)(lev) := f(i+8);
                xsave(i)(lev) := x(i+8);
        }
        --
        -- Assemble left hand elements for immediate use.
        --
        qprev := qleft;
        for i in 1..8 repeat {
                j := -i;
                f(2*j+18) := f(j+9);
                x(2*j+18) := x(j+9);
        }
        goto L30;
        --
        -- *** Stage 6 ***   Trouble section
        -- Number of function values is about to exceed limit.
        --
@L60    nofin := 2*nofin;
        levmax := levout;
        flag := flag + (b - x0)/(b - a);
        goto L70;
        --
        -- Current level is levmax.
        --
@L62    flag := flag + 1;
        --
        -- *** Stage 7 ***   Interval converged
        -- Add contributions into running sums.
        --
@L70    result := result + qnow;
        errest := errest + esterr;
        cor11 := cor11 + qdiff / 1023.0;
        --
        -- Locate next interval.
        --
```

```
@L72    if nim = 2*(nim quo 2) then goto L75;
        nim := nim quo 2;
        lev := lev - 1;
        goto L72;

@L75    nim := nim + 1;
        if lev <= 0 then goto L80;
        --
        -- Assemble elements required for the next interval.
        --
        qprev := qright(lev);
        x0 := x(16);
        f0 := f(16);
        for i in 1..8 repeat {
                f(2*i) := fsave(i)(lev);
                x(2*i) := xsave(i)(lev);
        }
        goto L30;
        --
        -- *** Stage 8 ***   Finalize and return
        --
@L80    result := result + cor11;
        --
        -- Make sure errest not less than roundoff level.
        --
        if errest = 0.0 then RETURN;
@L82    temp := abs(result) + errest;
        if temp ~= abs(result) then RETURN;
        errest := 2.0 * errest;
        goto L82;
}

-- Test with a well-known example: the result should be Pi.

import from R, I, TextWriter, Character, String;

f(x:R):R == 4.0/(x*x+1);

(result, errest, nofun, flag) := quanc8(f,0.0,1.0,0.00001,0.00001);

if zero? flag then {
  stdout << "result = " << result << newline;
  stdout << "error = " << errest << newline;
  stdout << "(after " << nofun << " function evaluations)" << newline;
}
else
  stdout << "error flag is " << flag << newline;
```

PART V

Reference

CHAPTER 22

Formal syntax

22.1 Source

Aldor source is a collection of lines containing a textual representation of a program.

Lines beginning with the character "`#`" are *system commands* and are not part of the program text.

22.1.1 Source inclusion

Source inclusion collects the source lines which make up a program. This process is controlled by the following system commands:

- `#include   "`*file-name*`"`
  `#reinclude "`*file-name*`"`
  These commands collect the lines from the named file. If a given file has already been included, then subsequent `include` commands for that file have no effect. A `reinclude` command always includes the file, whether or not it has already been included.
  The includer tries to find the file relative to the directory of the current source file and then in a sequence of user-specified and platform-specific directories.
- `#assert` *identifier*
  `#unassert` *identifier*
  These commands turn on or off named properties which may be tested for conditional source inclusion.
- `#if` *identifier*
  `#elseif` *identifier*
  `#else`
  `#endif`
  These commands provide conditional source inclusion.

- `#!` *text*
  `#`
  Lines are ignored if they begin with “`#!`” or begin with “`#`” and contain only white space.

22.1.2 Prepared source

The following commands allow source to be prepared by another program.

- `#line` *line-number* ["*file-name*"]
  The next line is recorded as occurring at the given line number in the named file. The first line of a file is line number 1. If the file name is missing, then the current file name is used.
- `#error` *text*
  An error is considered to have occurred at the position of the system command line and the given text is used as the message.

22.1.3 Other system commands

Other system commands control the environment:

- `#pile`
  `#endpile`
  These commands create a context in which indentation is significant. Braces may be used to revert to a normal state where indentation has no meaning, and it is possible to nest “`#pile`”/“`#endpile`” and “`{`”/“`}`” pairs. Closing “`#endpile`” lines at the end of the program may be omitted.
- `#library` *identifier* "*file-name*"
  This links an entry in the file system to an identifier in the source program. The identifier is treated as a defined constant with a domain value in the file scope. A specified sequence of directories is searched for the named library file. See “`#libraryDir`”.
- `#includeDir` "*directory-name*"
  `#libraryDir` "*directory-name*"
  The given directory is prepended to a sequence of directories to be searched for include files or libraries, respectively. The sequences are initialised to platform-specific sets, which may be augmented by environment variables or command line arguments.
- `#int` *options*
  This command controls the behaviour of the compiler when used interactively. See Section 18.2 for a complete description of available options.
- `#quit`
  The `quit` command causes the language processor to abandon the program. If Aldor is running in interactive mode (“`-g loop`”), then this command causes the termination of the interactive session.

22.2 Lexical structure

22.2.1 Characters

The *standard Aldor character set* contains the following 97 characters:

- the blank, tab and newline characters
- the Roman letters: a-z A-Z
- the digits: 0-9
- and the following special characters:

(	left parenthesis	)	right parenthesis
[	left bracket	]	right bracket
{	left brace	}	right brace
<	less than	>	greater than
,	comma	.	period
;	semicolon	:	colon
?	question mark	!	exclamation mark
=	equals	_	underscore
+	plus	-	minus (hypen)
&	ampersand	*	asterisk
/	slash	\	back-slash
'	apostrophe (quote)	`	grave (back-quote)
"	double quote	|	vertical bar
^	circumflex	~	tilde
@	commercial at	#	sharp
$	dollar	%	percent

Other characters may appear in source programs, but only in comments and string-style literals. Blank, tab and newline are called *white space* characters. All the special characters except quote, grave and ampersand are required for use in tokens. Grave and ampersand are reserved for future use.

22.2.2 The escape character

Underscore is used as an "escape" character, which alters the meaning of the following text. The nature of the change depends on the context in which the underscore appears.

An escaped underscore is not an escape character.

An escape character followed by one or more white space characters causes the white space to be ignored. The remainder of this section assumes that escaped white space has been removed from the source.

22.2.3 Tokens

The sequence of source characters is partitioned into *tokens*. The longest possible match is always used.

The tokens are classified as follows:

- the following language-defined *keywords*:

```
add      and       always    assert    break
but      catch     default   define    delay
do       else      except    export    extend
fix      for       fluid     free      from
generate goto      has       if        import
in       inline    is        isnt      iterate
let      local     macro     never     not
of       or        pretend   ref       repeat
return   rule      select    then      throw
to       try       where     while     with
yield

.   ,   ;   :   ::  :*  $   @
|   =>  +-> :=  ==  ==> '
[   ]   {   }   (   )
```

  The characters in a keyword cannot be escaped. That is, if a character is escaped, the token is not treated as a keyword.

- the following are not defined by the language but are *reserved words* for future use:

```
delay    fix      is       isnt      let      rule

(|  |)   [|  |]   {|  |}   `     &     ||
```

- the following set of definable *operators*:

```
by       case     mod      quo      rem

#        +        -        +-       ~        ^
*        **       ..       =        ~=       ^=
/  \     /\  \/   <   >    <=  >=   <<  >>   <-   ->
```

  The characters in an operator cannot be escaped.

- *identifiers*:

```
0
1
[%a-zA-Z][%?!a-zA-Z0-9]*
```

  Any non-white space standard character may be included in an identifer by escaping it. Thus “`a`”, “`_*`”, “`a_*`” and “`_if`” are all identifiers. The escape character is not part of the identifier so “`ab`” “`_a_b`” represent the same identifier. Identifiers are the only tokens for which the leading character may be escaped.

- *string-style literals*:

```
`"'[^"]*`"'
```

  An underscore or double quote may be included in a string-style literal by escaping it.

- *integer-style literals*:
  `[2-9]`
  `[0-9][0-9]+`
  `[0-9]+'r'[0-9A-Z]+`
  Escape characters are ignored in integer-style literals and so may be used to group digits.
- *floating point-style literals*:
  `[0-9]*'.'[0-9]+{[eE]{[+-]}[0-9]+}`
  `[0-9]+'.'[0-9]*{[eE]{[+-]}[0-9]+}`
  `[0-9]+[eE]{[+-]}[0-9]+`
  `[0-9]+'r'[0-9A-Z]*'.'[0-9A-Z]+{e{[+-]}[0-9]+}`
  `[0-9]+'r'[0-9A-Z]+'.'[0-9A-Z]*{e{[+-]}[0-9]+}`
  `[0-9]+'r'[0-9A-Z]+'e'{[+-]}[0-9]+`
  Escape characters are ignored in floating point-style literals and so may be used to group digits.
  Certain lexical contexts restrict the form of floats allowed. This distinguishes cases such as `sin 1.2` *vs* `m.1.2`. A floating point literal may not
  - begin with "`.`", unless the preceding token is a keyword other than "`)`", "`|)`", "`]`" or "`}`",
  - contain "`.`", if the preceding token is "`.`",
  - end with "`.`", if the following character is "`.`".
- *comments*:
  The two characters "`--`" and all characters up to the end of the line. Underscores are not treated as escape characters in comments.
- *documentation*:
  The two characters "`++`" and all characters up to the end of the line. Underscores are not treated as escape characters in documentation.
- *leading white space*:
  a sequence of blanks or tabs at the beginning of a line.
- *embedded white space*:
  a sequence of blanks or tabs not at the beginning of a line.
- *newline*:
  a newline character.
- *layout markers*:
  SETTAB *BACKSET* *BACKTAB*
  These do not appear in a source program but may be used to represent a linearized from of the token sequence.

Comments and embedded white space are always ignored, except as used to separate tokens. For example, "`abc`" is taken as one token but "`a b c`" is taken as three.

22.3 Layout

Normally page layout is not significant in an Aldor program. Within a "`#pile`"/"`#endpile`" pair, however, indendation and newlines have meaning, and are used to group collections of lines. Source within such a pair is in a *piling context*.

Indentation sensitivity may be turned off by enclosing source in a "`{`"/"`}`" pair. Within braces all white space — leading, embedded, and newlines — is ignored. This is a *non-piling context*. Piling and non-piling contexts may be nested.

The layout of a program in a piling context is understood by converting leading white space and newlines to special markers which are part of the language syntax.

This conversion follows the *linearization rules*:

- Blank lines are ignored.
- Consecutive lines indented the same ammount form a pile.
- The pile is enclosed in a *SETTAB*-*BACKTAB* pair if
  - the pile has more than one line, or
  - the pile has only one line and the last token of the line before the pile is "`then`", "`else`", "`with`" or "`add`".
- The pile lines are joined to form a single line. A *BACKSET* inserted between adjacent pair of lines, unless
  - the first contains only comments or documentation, or
  - the first ends with "`(`", "`[`", "`{`" or "`,`", or
  - the second begins with "`in`", "`then`", "`else`", "`)`", "`]`" or "`}`".
- A line is joined to the previous line if it is indented with respect to it, forming a new single line.

These rules are applied from the most indented lines back out to the least indented lines.

22.4 Grammar

This section presents the grammar used by the Aldor compiler. After expanding parameterized rules, this grammar is conflict-free and LALR(1).

Declarative expressions

Goal : CurlyContents(*Labeled*)

Expression : enlist1a(*Labeled*, “`;`”)

Labeled : *Comma* | *Declaration* | “`@`” *Atom* [*Labeled*]

Declaration:
  “`macro`” *Sig*
  | “`extend`” *Sig*
  | “`local`” *Sig*
  | “`free`” *Sig*
  | “`fluid`” *Sig*
  | “`default`” *Sig*
  | “`define`” *Sig*
  | “`fix`” *Sig*
  | “`inline`” [*Sig*] [*FromPart*]
  | “`import`” [*Sig*] [*FromPart*]
  | “`export`” [*Sig*] | “`export`” [*Sig*] *ToPart* | “`export`” [*Sig*] *FromPart*

ToPart : “`to`” *Infixed*

FromPart : “`from`” enlist1(*Infixed*, “`,`”)

Sig : *DeclBinding* | *Block*

DeclPart : “`:`” *Type* | “`:*`” *Type*

Comma : enlist1(*CommaItem*, “`,`”)

CommaItem :
  Binding(*AnyStatement*)
  | Binding(*AnyStatement*) “`where`” *CommaItem*

DeclBinding : BindingR(*InfixedExprsDecl*, *AnyStatement*)

InfixedExprsDecl : *InfixedExprs* | *InfixedExprs* *DeclPart*

InfixedExprs : enlist1(*InfixedExpr*, “`,`”)

Binding(E) : BindingL(*Infixed*, E)

BindingL(R, L):
  L
  | R “`:=`” BindingL(R, L) | R “`==`” BindingL(R, L)
  | R “`==>`” BindingL(R, L) | R “`+->`” BindingL(R, L)

BindingR(R, L):
  R

| R ":=" Binding(L) | R "==" Binding(L)
| R "==>" Binding(L) | R "+->" Binding(L)

Control flow

AnyStatement:
"`if`" *CommaItem* "`then`" Binding(*AnyStatement*)
| Flow(*AnyStatement*)

BalStatement : Flow(*BalStatement*)

Flow(X):
Collection
| "`if`" *CommaItem* "`then`" Binding(*BalStatement*) "`else`" Binding(X)
| *Collection* "`=>`" Binding(X)
| *Iterators* "`repeat`" Binding(X)
| "`repeat`" Binding(X)
| "`try`" Binding(*AnyStatement*) "`but`" [*Cases*] AlwaysPart(X)
| "`select`" Binding(*AnyStatement*) "`in`" *Cases*
| "`do`" Binding(X)
| "`delay`" Binding(X)
| "`generate`" *GenBound* Binding(X)
| "`assert`" Binding(X)
| "`iterate`" [*Name*]
| "`break`" [*Name*]
| "`return`" [*Collection*]
| "`yield`" Binding(X)
| "`except`" Binding(X)
| "`goto`" *Id*
| "`never`"

GenBound : *Nothing* | "`to`" *CommaItem* "`of`"

Cases : Binding(*Collection*)

AlwaysPart(X) : "`always`" Binding(X) | *Nothing*

Collection : *Infixed* | *Infixed Iterators*

Iterators : *Iterators1*

Iterators1 : *Iterator* | *Iterators1* Iterator

Iterator : "`for`" *ForLhs* "`in`" *Infixed* [*SuchthatPart*] | "`while`" *Infixed*

ForLhs : *Infixed* | "`free`" *Infixed* | "`local`" *Infixed* | "`fluid`" *Infixed*

SuchthatPart : "`|`" *Infixed*

Infixed expressions

Infixed : *InfixedExpr* | *InfixedExpr DeclPart* | *Block*

InfixedExpr : E11(*Op*) | *E3*

E3 : *E4* | *E3* "`and`" *E4* | *E3* "`or`" *E4* | *E3 LatticeOp E4*

E4 : *E5* | *E4* "`has`" *E5* | *E4 RelationOp E5* | *RelationOp E5*

E5 : *E6* | *E5 SegOp* | *E5 SegOp E6*

E6 : *E7* | *E6* *PlusOp* *E7* | *PlusOp* *E7*

E7 : *E8* | *E7* *QuotientOp* *E8*

E8 : *E9* | *E8* *TimesOp* *E9*

E9 : E11(*E12*) | E11(*E12*) *PowerOp* *E9*

E11(X):
  X | E11(X) "`::`" *E12* | E11(X) "`@`" *E12* | E11(X) "`pretend`" *E12*

Type : E11(*E12*)

E12 : *E13* | *E13* *ArrowOp* *E12*

E13 : *E14* | *E14* "`$`" *QualTail*

QualTail : *LeftJuxtaposed* | *LeftJuxtaposed* "`$`" *QualTail*

OpQualTail : *Molecule* | *Molecule* "`$`" *OpQualTail*

E14:
  E15 | *E14* "`except`" *E15*
  | [*E14*] "`with`" *DeclMolecule* | [*E14*] "`add`" *DeclMolecule*

E15 : *Application*

Infixed operators

Op:
  ArrowOp | *LatticeOp* | *RelationOp* | *SegOp*
  | *PlusOp* | *QuotientOp* | *TimesOp* | *PowerOp*

NakedOp:
  ArrowTok | *LatticeTok* | *RelationTok* | *SegTok*
  | *PlusTok* | *QuotientTok* | *TimesTok* | *PowerTok*

ArrowOp: QualOp(*ArrowTok*)
LatticeOp: QualOp(*LatticeTok*)
RelationOp: QualOp(*RelationTok*)
SegOp: QualOp(*SegTok*)
PlusOp: QualOp(*PlusTok*)
QuotientOp: QualOp(*QuotientTok*)
TimesOp: QualOp(*TimesTok*)
PowerOp: QualOp(*PowerTok*)

ArrowTok: "`->`" | "`<-`"

LatticeTok: "`\/`" | "`/\`"

RelationTok:
  "`=`" | "`~=`"| "`^=`" | "`>=`" | "`>`" | "`>>`" | "`<=`" | "`<`" | "`<<`" |
  "`is`" | "`isnt`" | "`case`"

SegTok: "`..`" | "`by`"

PlusTok: "`+`" | "`-`" | "`+-`"

QuotientTok: "`mod`" | "`quo`" | "`rem`"

TimesTok: "*" | "/" | "\"

PowerTok: "**" | "^"

Juxtaposed expressions

The expressions "a b", "a.b", and "a(b)" have the same semantics, but different grouping. The following rules provide desired precedence. Juxtaposition "a b" is looser than "." and "a(b)", and associates the opposite way:

```
A B C D                    as (.(.(.)))
f(a).2(b)(c).x.y.(d).(e)   as (((.).).)
```

This allows both the nested function application "sin cos x" and the data access "T.x.first.tag" to be written naturally.

Application : *RightJuxtaposed*

RightJuxtaposed : Jright(*Molecule*)

LeftJuxtaposed : Jleft(*Molecule*)

Jright(H) : Jleft(H) | Jleft(H) Jright(*Atom*) | "not" Jright(*Atom*)

Jleft(H) :
  H | "not" *BlockEnclosure* | Jleft(H) *BlockEnclosure*
  | Jleft(H) "." *BlockMolecule*

Primary expressions

Molecule : *Atom* | *Enclosure*

Enclosure : *Parened* | *Bracketed* | *QuotedIds*

DeclMolecule : [*Application*] | *Block*

BlockMolecule : *Atom* | *Enclosure* | *Block*

BlockEnclosure : *Enclosure* | *Block*

Block : Piled(*Expression*) | Curly(*Labeled*)

Parened : "{" "}" | "{" *Expression* "}"

Bracketed : "[" "]" | "[" *Expression* "]"

QuotedIds : "'" "'" | "'" *Names* "'"

Names : enlist1(*Name*, ",")

Terminals

Atom : *Id* | *Literal*

Name : *Id* | *NakedOp*

Id : TK_Id | TK_Blank | "#" | "~"

Literal : TK_Int | TK_Float | TK_String

Meta-rules

Nothing :

QualOp(op) : op | op "$" *OpQualTail*

[E] : *Nothing* | E

Documentation

Doc(E) : *PreDocument* E *PostDocument*

PreDocument : *PreDocumentList*

PostDocument : *PostDocumentList*

PreDocumentList : *Nothing* | TK_PreDoc *PreDocumentList*

PostDocumentList : *Nothing* | TK_PostDoc *PostDocumentList*

Separated lists

The rule enlist1 provides lists with separators between elements, *e.g.* "`x , y, z`".

enlist1(E, Sep) : E | enlist1(E, Sep) Sep E

The rule enlist1a provides lists within which separators may be repeated, *e.g.* "`x ; ; y ; z`". Any number of separators may follow the last element.

enlist1a(E, Sep) : E | enlist1a(E, Sep) Sep E | enlist1a(E, Sep) Sep

Blocks

Piled(E) : KW_SetTab PileContents(E) KW_BackTab

Curly(E) : "`{`" CurlyContents(E) "`}`"

PileContents(E):
  Doc(E)
  | PileContents(E) KW_BackSet Doc(E)
  | error KW_BackSet Doc(E)

CurlyContents(E) : CurlyContentsList(E)

CurlyContentsList(E) : CurlyContent1(E)
  | CurlyContent1(E) CurlyContentB(E)

CurlyContent1(E) : *Nothing* | CurlyContent1(E) CurlyContentA(E)

CurlyContentA(E):
  CurlyContentB(E) "`;`" *PostDocument*
  | error "`;`" *PostDocument*

CurlyContentB(E) : *PreDocument* E *PostDocument*

CHAPTER 23

Command line options

The `aldor` command has the following general form:

`aldor` *options* *file1* *file2* ...

This compiles the files one after the other, each in a fresh environment. Depending on the particular command line options given by the user, the files resulting from the compilations may be combined or run.

The options are case-insensitive so "`-G INTERP`" is treated the same way as "`-g interp`".

If the environment variable "`ALDORARGS`" is defined, its contents are handled as options before the ones on the command line.

23.1 File types

Files with the following type extensions are accepted on the command line:

`.as` *Aldor source.* If a file name has no type extension, then it is treated as if it were a "`.as`" file.

`.ai` *Included Aldor source.* This is a file with all "`#include`" and "`#if`" statements processed. This sort of file is produced by running the compiler with a "`-Fai`" option.

`.ap` *Parsed Aldor source.* This is a file with the program in an S-expression syntax. This is the easiest way to have the Aldor compiler process programs generated by a Lisp program. This sort of file is produced by running the compiler with a "`-Fap`" option.

`.fm`	*Foam source.* "Foam" is an acronym for "First Order Abstract Machine". The Aldor compiler produces Foam as its intermediate code. An "`fm`" file contains Foam code in S-expression syntax. This sort of file is produced by running the compiler with a "`-Ffm`" option.
`.ao`	*Machine-independent object file.* This is the result of compiling an Aldor source file. It contains type information, documentation, Foam code, symbol table information, and so on. Characters are internally represented in ASCII form, and floating point numbers are represented in a transportable format guaranteed not to loose significance or exponent range on any platform.
`.al`	*Archive of machine-independent object files.* This is treated as a homogeneous aggregated library by the compiler. The file is laid out as a Unix-style archive on all platforms. Thus both "`.ao`" and "`.al`" files may be moved from machine to machine.
obj	*Object file.* This is a platform-specific object file. These are named in the usual way for the platform. For example on Unix these files have extension "`.o`", while on DOS they have extension "`.obj`" and on CMS they have extension "`TEXT`".
arch	*Archive of object files.* The name, contents, and format of these archives is determined by the platform. For example on Unix this would be a "`.a`" file containing "`.o`" files.

23.2 General options

`-V`	Run verbosely, giving compilation information.
`-D` *id*	Add global assertion as "`#assert` *id*".
`-U` *id*	Remove global assertion as "`#unassert` *id*".
`-A` *fn*	Read command line options from response file *fn*.
`-K` *n*	Compile only the first *n* files (0-9).
`--`	Treat remaining arguments as input files.
`-H ...`	Help.
`-B` *dir*	Use *dir* as the base directory for Aldor system files.
`-I` *dir*	Search *dir* for additional include files.
`-Y` *dir*	Search *dir* for additional libraries.
`-R` *dir*	Put the resulting files in directory *dir*.

`-L ...`	Use the given library.
`-F ...`	Indicate which output files are to be generated.
`-E` *fn*	Specify the main entry point.
`-G`	Run the program.
`-O`	Standard optimisations.
`-Q ...`	Select code optimisations.
`-Z ...`	Debugging and profiling options.
`-C ...`	Control C generation.
`-S ...`	Control Lisp generation.
`-M`	Control compiler messages.
`-W`	Developer options.

23.3 Help options

`-H elp`	Brief, general help.
`-H all`	Help about *all* options.
`-H files`	Help about input file types.
`-H options`	Help about summary of options.
`-H [A|args]`	Help about argument gathering options.
`-H [H|help]`	Help about help options.
`-H dir`	Help about directory and library options.
`-H [F|fout]`	Help about output file options.
`-H [G|go]`	Help about execution options.
`-H [O|Q|optimise]`	Help about optimisation options.
`-H [Z|debug]`	Help about debugging options.
`-H C`	Help about C code generation options.
`-H [S|lisp]`	Help about Lisp code generation options.
`-H [M|message]`	Help about message options.
`-H [W|dev]`	Help about developer options.

23.4 Argument gathering options

`-A` *fn*	Read command line options from response file *fn*.
`-K` *n*	Use only the first *n* files (0-9). The remaining arguments are given to the Aldor program, if run with "`-G`".
`--`	Treat remaining arguments as input files, even if they begin with "`-`". If the environment variable "`ALDORARGS`" is defined, its contents are handled as options before the command line.

23.5 Directories and libraries options

`-I` *dir*	Search *dir* for include files. The environment variable "`INCPATH`" provides default locations.
`-Y` *dir*	Search *dir* for additional libraries. The environment variable "`LIBPATH`" provides default locations.
`-R` *dir*	Put the resulting files in directory *dir*. The default is the current directory.
`-B` *dir*	Use *dir* as the base directory for Aldor system files. If the "`-B`" option is omitted, then the base directory must be given by the environment variable "`ALDORROOT`".
`-L` *fn*	Use the library given by file name *fn*. An alphanumeric *fn* is a short form for "`lib`*fn*`.a`".
`-L` *id*=*fn*	Same as "`-L` *fn*", but the source name *id* is associated with the library.

23.6 Generated file options

The "`-F`" option indicates which output files are to be generated. Options marked "(*)" cause one file of the given type to be generated for each input file, whilst those marked "(1)" cause one file to be generated for the entire compilation.

`-F ai`	(*) Source after all `#include` statements have been processed (from "`.as`")
`-F ax`	(*) Macro-expanded parse tree (from "`.as`")
`-F asy`	(*) Symbol information
`-F ao`	(*) Machine-independent object file
`-F fm`	(*) Foam code
`-F lsp`	(*) Lisp code
`-F c`	(*) C code
`-F o`	(*) Object file

`-F x`	(1) Executable file
`-F aldormain`	(1) Generate "`aldormain.c`" containing function "`main`".

If no "`-F`" or "`-G`" option is given, then "`-Fao`" is assumed.

If the parameter *fn* is given, it is used as the name of the corresponding output file.

The compiler will not overwrite a C or Lisp file it did not generate.

23.7 Execution options

`-G run`	Compile the program to machine code, and then run it. The executable file is removed afterwards unless the "`-Fx`" option is present.
`-G interp`	Translate the program to Foam code, and then run it in interpreted mode. The "`.ao`" file is removed afterwards unless the "`-Fao`" option is present.
`-G loop`	Run interactively, interpreting each expression typed by the user. With this option, the file "`aldorinit.as`" is used for initialisation.
`-E` *fn*	Use the input file "*fn*`.*`" as the main entry point. The default is the first file. "`-E`" is useful only with "`-Fx`", "`-Grun`" or "`-Ginterp`".

23.8 Optimisation options

Combinations can be used, *e.g.* "`-OQno-cc`" or "`-Qno-all -Qcc`". The default is "`-Qcfold -Qdeadvar -Qpeep`". The option "`-Qcc`" may exclude "`-[gp]`" on some platforms.

`-O`	Optimise. This is equivalent to "`-Q2`".
`-Q` *n*	Select a level of code optimisation. (default is "`-Q1`").
`-Q` *opt*	Turn on an optimisation *opt* from the list below.
`-Q no-`*opt*	Turn off an optimisation *opt* from the list below.

		Q0	Q1	Q2	Q3
`-Q all`	All supported optimisations.				X

`-Q inline`	Allow open coding of functions.			X	X
`-Q inline-all`	Allow open coding of any functions.				X
`-Q inline-limit=`n	Set maximum acceptable increase in code size from inlining to n.	1.0	1.0	5.0	6.0
`-Q cfold`	Evaluate non floating point constants at compile time.		X	X	X
`-Q ffold`	Evaluate floating point constants at compile time.			X	X
`-Q hfold`	Determine, where possible, the run time hash code which domains will have, reducing the number of domain instantiations and speeding up cross-file look-ups.		X	X	X
`-Q peep`	Local “peep-hole” optimisations.		X	X	X
`-Q deadvar`	Eliminate unused variables and values.		X	X	X
`-Q emerge`	Combine lexical levels and records.			X	X
`-Q cprop`	Copy propagation.			X	X
`-Q cse`	Common sub-expression elimination.			X	X
		Q0	Q1	Q2	Q3
`-Q flow`	Simplify computed tests and jumps.			X	X
`-Q cast`	Reduce the number of casts.			X	X

`-Q cc`	Use the C compiler's optimiser.	X	X

Combinations can be used, *e.g.* "`-Q3 -Qno-ffold`". The option "`-Qcc`" may exclude "`-Z`" on some platforms.

23.9 Debug options

`-Z db`	Generate debugging information in object files.
`-Z prof`	Generate profiling code in object files.

23.10 C code generation options

Control the behaviour of "`-F c`", the option for generating C code.

`-C standard`	Generate ANSI/ISO standard C (the default).
`-C old`	Generate old (Kernighan & Ritchie) C. The options "`-Cstandard`" and "`-Cold`" are mutually exclusive.
`-C comp=`*name*	Use *name* instead of the default C compiler driver `unicl`. Use "`-v`" to see how the driver is invoked.
`-C link=`*name*	Use *name* instead of the default linker driver `unicl`. Use "`-v`" to see how the driver is invoked.
`-C cc=`*name*	Use *name* instead of the default C compiler and linker driver `unicl`. Without this option, the environment variable "`CC`" is tried. Use "`-v`" to see how the driver is invoked.
`-C go=`*name*	Use *name* to run the output of the linker. Without "`-Cgo=`", the environment variable "`CGO`" is tried. Use "`-v`" to see how the driver is invoked. Most implementations don't need this.
`-C args=`*opts*	Pass *opts* as options to the C compiler and linker. Use "`-v`" to see how the driver is invoked.
`-C smax=`*n*	Attempt to put no more than *n* statements in each file, if necessary by splitting the generated file into: "`name.h`", "`name.c`", "`name001.c`", "`name002.c`", etc. Using "`-C smax=0`" turns off file splitting. (default: "`smax=2000`")
`-C idlen=`*n*	Set the maximum length of C identifiers to be *n*. Using "`-C idlen=0`" turns off identifier truncation. (default: "`idlen=32`")

`-C [no-]idhash`	(Do not) use hash codes in global C identifiers. (default: "`idhash`")
`-C [no-]lines`	Preserve Aldor source line numbers in generated C. (default: no-lines)
`-C sys=`*name*	Pass the option "`-Wsys=`*name*" to the C compiler and linker. This option is interpreted by `unicl` to select a group of options defined in its configuration file "`aldor.conf`". It also prepends to the list of library directories a modified list in case there is a special implementation for the particular name. This is of great help when making libraries for specific flavours of CPU.
`-C runtime=`*id1*`,..`	Select a list of libraries "`lib`*id1*`.a`", "`lib`*id2*`.a`" etc. that implement the runtime system. Default is "`foam`".
`-C fortran`	Pass "`-Wfortran`" to link driver `unicl` which enables fortran options such as fortran runtime libraries.
`-C lib=`*name*	Link with the library "`lib`*name*`.a`"

23.11 Lisp code generation options

Control the behaviour of "`-F lsp`".

`-S common`	Produce Common Lisp code (the default).
`-S standard`	Produce Standard Lisp code.
`-S scheme`	Produce Scheme code. The options "`-Scommon`", "`-Sstandard`" and "`-Sscheme`" are mutually exclusive.
`-S ftype=`*ft*	Use *ft* as the file extension for the generated lisp file (default: "`-Sftype=lsp`").

23.12 Message options

`-M emax=`*n*	Stop after *n* error messages. (default: 10)
`-M db=`*fn*	Use *fn* as the message database. (default: no-db)
`-M` *msgname*	Turn on warning or remark named *msgname*. Use "`-Mname`" (described below) to find out the names of messages.

`-M no-`*msgname*	Turn off warning or remark named *msgname*.				
`-M` *n*	Control how much detail is given. (default: "`-M2`")				
`-M no-`*opt*	Turn of "`-M` *opt*".				
		M0	M1	M2	M3
`-M warnings`	Display warnings.	N	Y	Y	Y
`-M source`	Display the program lines for messages.	N	N	Y	Y
`-M details`	Display details.	N	N	Y	Y
`-M notes`	Display cross reference notes.	N	N	Y	Y
`-M remarks`	Display remarks.	N	N	N	Y
`-M sort`	Sort messages by source position.	Y	Y	Y	Y
`-M mactext`	Point to macro text, rather than use.	Y	Y	Y	Y
`-M abbrev`	Abbreviate types in messages.	Y	Y	Y	Y
`-M human`	Human-oriented format.	Y	Y	Y	Y
`-M name`	Show the name of each message as well as the message itself.	N	N	N	N
`-M antiques`	Display warnings for old-style code.	N	N	N	N
`-M preview`	Display messages as they occur.	N	N	N	N
`-M inspect`	Use interactive inspector for errors.	N	N	N	N

23.13 Developer options

Developer options (subject to change):

`-W check`	Turn on internal safety checks.
`-W runtime`	Produce code suitable for the runtime system.
`-W nhash`	Assume all exported types have constant hash-codes.
`-W loops`	Always inline generators when possible.
`-W missing-ok`	Do not stop the compiler if an export is missing in a domain.
`-W audit`	Set maximum Foam auditing level.
`-W trap`	Trap failure exits (for debugging Aldor).
`-W gc`	Garbage collect as needed (if gc is available).
`-W gcfile`	Garbage collect after each file (if gc is available).
`-W sexpr`	Run a read-print loop.
`-W seval`	Run a read-eval-print loop.
`-W test=`*name*	Run compiler self-test *name*. Use the option "`-W test+show`" to see a list of self-tests.
`-W D+`*name*	Turn on debug hook *name*. Use "`-W D+show`" to see a list of debug hooks.
`-W D-`*name*	Turn off debug hook *name*. Use "`-W D+show`" to see a list of debug hooks.
`-W Tapdrgst0+`*ph*	Phase tracing: Several phases can be given as "`ph1+ph2+ph3`" or "`all`". Several options can be given at once, *e.g.* "`-WTags+all`". Compile a file with "`-v`" to see the phase abbreviations. The remaining options give more details about the trace code letters.
`-WTa+`*ph*	Announce entry to *ph*.
`-WTp+`*ph*	Pretty print result of *ph*.
`-WTd+`*ph*	Print debug information for *ph*.
`-WTr+`*ph*	Show result of *ph*.
`-WTg+`*ph*	Garbage collect after *ph*.
`-WTs+`*ph*	Storage audit after *ph*.
`-WTt+`*ph*	Terminate after *ph*.
`-WT0+`*ph*	Ignore earlier "`-WT`" option for *ph*.

23.14 Environment variables

Sometimes there are certain aspects of the compiler's behaviour which you may wish to change for most of your compilations.

Most operating systems have some notion of environment variables and the Aldor compiler checks a number of these to guide its actions. None of these needs to be defined, but if they are the Aldor compiler will use them.

`ALDORROOT` — This gives a directory under which the compiler will find its own include files, libraries, *etc.*

`ALDORARGS` — This provides options to the compiler which are treated before those appearing on the command line.

`INCPATH` — This gives the compiler additional places to search for include files. The syntax is according to the operating system's conventions. For example, on Unix a suitable initialisation could be:

```
INCPATH=/home/jane/include:.:/usr/local/include
```

This has the same effect as using the command line options

```
-I/home/jane/include -I. -I/usr/local/include
```

`LIBPATH` — This gives the compiler additional places to search for libraries.

`CC` — This gives the Aldor compiler the name of a C compiler which it should call to generate and link object code. The default is `unicl` – a program provided in the Aldor distribution to convert C compile command lines to the native system syntax.

`CGO` — This gives the Aldor compiler the name of a loader which it should call to run an executable program it has generated. On most platforms no explicit loader is needed.
Sometimes you will wish to specify "`CC`" and "`CGO`" together. For example on DOS using the "`djgpp`" port of GCC, a useful combination is

```
set CC=gcc
set CGO=go32
```

CHAPTER 24

The `unicl` driver

The `unicl` program, which is supplied in the distribution and can be found in `$ALDORROOT/bin`, is a configurable driver for the C compiler and linker. It is used whenever the `aldor` options `-Fo` or `-Fx` are specified.

The `aldor` option `-v` can be used to show how `unicl` is invoked and how it calls the underlying C compiler. Invoking `unicl` on its own will display some useful information about the options it understands. These include some options commonly found in C compilers such as

- `-O` : Optimize.
- `-c` : Compile only, do not link.
- `-g` : Include debug information in generated objects.
- `-o` *name* : Name the executable.
- `-p` : Include profiling code.
- `-D` *def* : Define a macro.
- `-U` *undef* : Undefine a macro.
- `-I` *dir* : Specify directory for included files.
- `-L` *dir* : Specify directory for linked libraries.
- `-l` *lib* : Specify libraries to link.

Additionally, `unicl` understands the following options

- `-Wh` : Show help information.
- `-Wv` : Be verbose.
- `-Wv=`*n* : Set verbosity level (currently 0-4).
- `-Wn` : Do not execute generated commands.

An extra set of options act as flags for certain non-default `unicl` behaviour. These are

- `-Wstdc` : Use an ANSI C compiler.
  This is passed from `aldor` unless `aldor -Cold` is specified.

`-Wshared` : Produce a shared object.

`-Wfortran` : Link in FORTRAN code.
This is passed from `aldor` when `aldor` option `-Cfortran` is specified.

`-Wfnonstd` : Enable non-standard (non-IEEE) floating point.
This is passed from `aldor` when `aldor -Q4` or `-Qcc-fnonstd` is specified.

The `unicl` program will also look for options in the `UNICL` environment variable and will prepend them to the command line options if the variable exists.

The `unicl` program itself does not have any knowledge about the underlying C compiler for each platform. All the necessary information is read by `unicl` from an ASCII configuration file. By default, this file is searched for in `$ALDORROOT/include/aldor.conf` but the

`-Wconfig=`*file* : Specify location of configuration file.

option can be used to select an alternative. The configuration file consists of named sections introduced by an identifier enclosed in square brackets. Each section consists of assignments of the form key=value. The options

`-Wsys=`*name* : Specify which section of the configuration file to use.

`-Wv=2` : Show which section is being used.

set and show the section selected. When `aldor` calls `unicl` it automatically [1] specifies the appropriate section of the configuration file. The advantage of this method is that all the details for every implementation of Aldor can be kept in one place. It also means that the compiler user has complete control over the back end C compiler and linker. It is a simple matter, for instance to introduce a new section with a modified name, populate it with some variations on the original values and give the `aldor` option `-Csys=newname`.

These are the meaningful keys in the configuration file.

- The name of the program to call for compiling and linking

  `cc-name` : compiling only, non-ANSI C

  `link-name` : linking, non-ANSI C

  `std-cc-name` : compiling only, ANSI C

  `std-link-name` : linking, ANSI C

- The flag to use when `-g` (debugging) is given

  `cc-debug` : compiling only, non-ANSI C

  `link-debug` : linking, non-ANSI C

  `std-cc-debug` : compiling only, ANSI C

[1] Currently, it does this via the `UNICL` environment variable so you need to give the `aldor` option `-Cargs=-Wv=3` to see that. The `unicl` option `-Wstdc` is also passed that way unless you specify `aldor` option `-Cold`.

`std-link-debug` : linking, ANSI C

- The flag to use when `-p` (profiling) is given

`cc-profile` : compiling only, non-ANSI C
`link-profile` : linking, non-ANSI C
`std-cc-profile` : compiling only, ANSI C
`std-link-profile` : linking, ANSI C

- The flags to use when `-O` (optimise) is given (without the option `-Wfnonstd`)

`cc-optimize` : compiling only, non-ANSI C
`link-optimize` : linking, non-ANSI C
`std-cc-optimize` : compiling only, ANSI C
`std-link-optimize` : linking, ANSI C

- The flags to use when `-O` and `-Wfnonstd` (non-IEEE optimisation) are given

`cc-non-std-float` : compiling only, non-ANSI C
`link-non-std-float` : linking, non-ANSI C
`std-cc-non-std-float` : compiling only, ANSI C
`std-link-non-std-float` : linking, ANSI C

- The flags to use in any case. These will appear before any specified files.

`cc-opts` : compiling only, non-ANSI C
`link-opts` : linking, non-ANSI C
`std-cc-opts` : compiling only, ANSI C
`std-link-opts` : linking, ANSI C

- The flags to use in any case. These will appear at the end of the command.

`cc-post` : compiling only, non-ANSI C
`link-post` : linking, non-ANSI C
`std-cc-post` : compiling only, ANSI C
`std-link-post` : linking, ANSI C

- The flags to use in any case. These will appear after filenames but before any library specification only at the link step.

`link-twixt` : linking, non-ANSI C
`std-link-twixt` : linking, ANSI C

- The flags to use when linking libraries.

`library` : The flag for linking libraries (commonly `-l`).
`library-sep` : Do we need a space between the flag and the name of the library (true/false).

`lib-extra` : A space delimited list of libraries to be linked.
Example: `nsl socket`.

`libpath` : The flag for specifying library directories (commonly `-L`).

`libpath-sep` : Do we need a space between the flag and the name of the library directory (true/false).

`expand-libs` : Do we turn `-llib` options into complete file arguments (true/false).

`lib-ext` : The file extension for libraries (commonly `a`).

`lib-default-path` : A space delimited list of the directories to be searched for libraries.
Example: `/usr/local/lib /usr/X11/lib`.

- The flags to use when specifying include file directories.

`include` : The flag for specifying include file directories (commonly `-I`).

`include-sep` : Do we need a space between the flag and the name of the include file directories (true/false).

`include-default-path` : A space delimited list of directories to be searched for include files.
Examples: `/usr/local/include /usr/X11/include`.

- The flags to use when defining or undefining macros.

`define` : The flag for defining a macro (commonly `-D`).

`define-sep` : Do we need a space between the flag and the name of the macro (true/false).

`undefine` : The flag for undefining a macro (commonly `-U`).

`undefine-sep` : Do we need a space between the flag and the name of the macro (true/false).

- The flag to use when specifying name of linked executable.

`output-name` : The flag for specifying the name of the executable (commonly `-o`).

`output-name-sep` : Do we need a space between the flag and the name (true/false).

- Miscellaneous keys

`compile-only` : The flag that specifies compilation only (commonly `-c`).

`debug-profile-ok` : Can we specify debugging and profiling at the same time (true/false).

`debug-optimize-ok` : Can we specify debugging and optimizing at the same time (true/false).

`fortran-libraries` : A string which will be passed verbatim to the linker when `-Wfortran` is specified (see Section 20.5). For example: `-lfort -lf77`

- Keys that `aldor` looks for

`generate-stdc` : Do we generate ANSI C (true/false).

`fortran-cmplx-fns` : The scheme to use when dealing with FORTRAN functions which return complex numbers. The possible values are `string`, `return-void`, `return-struct`, and `disallowed`. See Section 20.5.

`fortran-naming-scheme` : Which scheme to use when resolving external FORTRAN names. The possible values are `underscore` `no-underscore` `underscore-bug`. See Section 20.5.

The special key `inherit` takes as value a section name and directs `unicl` to look in the named section if a key is not found in the current one. The special character `$` prepended to a key name stands for the value of that key. You can introduce your own keys for convenience and refer to them using `$`. The following conditions are provided

`compile` the program is compiling C code (one of 'link' and 'compile' will be true)

`link` the program is linking its arguments to an executable

`stdc` `-Wstdc` was specified

`shared` `-Wshared` was specified

`optimize` `-O` was specified

`nonstdfloat` `-Wnonstdfloat` was specified

`stdfloat` `-Wnonstdfloat` was not specified

`profile` `-p` was specified.

You can use the conditions when specifying values with this form:

```
{\tt \$?condition w1 w2 w3 : w4 ;}
```

meaning 'If condition is true then `w1 w2 w3` else `w4`'. Conditional forms can be nested.

CHAPTER 25

Compiler messages

This chapter lists the messages produced by the Aldor compiler. Each message has a *name* and an associated *text*. For example, one message has the name "`ALDOR_E_TinOpMeans`" and the associated text "`Operator has %d possible types.`"

The name of this message has four parts:

`ALDOR`	indicates this message is for the Aldor compiler,
`E`	indicates this is an *error* message,
`Tin`	indicates the message arises in type inference,
`OpMeans`	identifies the particular message.

A number of letters may occur in the place of "`E`" above. The letters and their meanings are:

`F`	fatal errors — compiler stops
`E`	soft errors — compiler keeps going
`W`	warnings
`R`	remarks

`H`	help
`M`	interactive loop messages
`N`	notes — cross references to other source lines
`D`	details for another message
`P`	punctuation

The name can be used to enable or disable the display of a specific warning or a remark. See Section 16.7 on page 170 for details.

We show below the name and the text of most of the compiler messages. The only messages we have omitted are those which give the command line help:

```
ALDOR_H_HelpCmd               ALDOR_H_HelpFileTypes
ALDOR_H_HelpOptionSummary     ALDOR_H_HelpHelpOpt
ALDOR_H_HelpArgOpt            ALDOR_H_HelpDirOpt
ALDOR_H_HelpFileOpt           ALDOR_H_HelpGoOpt
ALDOR_H_HelpOptimOpt          ALDOR_H_HelpDebugOpt
ALDOR_H_HelpCOpt              ALDOR_H_HelpLispOpt
ALDOR_H_HelpMsgOpt            ALDOR_H_HelpDevOpt
ALDOR_H_HelpMenuPointer
```

This information is given in chapter 23.

Within the text of the messages, items such as "`%s`" and "`%d`" have strings and numbers substituted into them before the message is actually used. For example, the message with the text "`Operator has %d possible types.`" might appear as "`Operator has 4 possible types.`" when it is actually displayed in a given context.

`ALDOR_E_ExplicitMsg`	%s
`ALDOR_N_ExplicitMsg`	%s
`ALDOR_N_Here`	
`ALDOR_F_CmdBadOption`	Improper use of '%s' option. Type '%s -help' for help.
`ALDOR_F_CmdNoOption`	'%s' is not an option. Type '%s -help' for help.
`ALDOR_F_CmdCantUseEntry`	Could not use file '%s' as the main file.
`ALDOR_F_CmdNoOutputDir`	The specified output directory does not exist.
`ALDOR_W_CmdFunnyEntry`	Bogus main file name '%s' ignored.
`ALDOR_W_DisableNotKeyword`	Unable to enable or disable '%s': not a keyword.
`ALDOR_W_FtnVarStringRet`	You cannot receive variable-length strings from Fortran functions. Use FixedString() instead.
`ALDOR_W_FtnNotFtnArg`	Unrecognised argument type for Fortran call.
`ALDOR_F_MsgTooManyErrors`	Too many errors (use '-M emax=n' or '-M no-emax' to change the limit).
`ALDOR_R_MsgCountMessages`	This file had %d errors, %d warnings and %d remarks.
`ALDOR_R_MsgAdviseDetails`	Use 'aldor -M<n>' to get more or less detail.
`ALDOR_R_MsgCondolences`	Sorry, your file did not compile.
`ALDOR_R_MsgCongratulations`	Congratulations, your file compiled!!
`ALDOR_P_MsgTagRemark`	(Remark)
`ALDOR_P_MsgTagWarning`	(Warning)
`ALDOR_P_MsgTagError`	(Error)
`ALDOR_P_MsgTagFatal`	(Fatal Error)
`ALDOR_P_MsgTagNote`	(Note %d)
`ALDOR_P_MsgPreview`	(Message Preview)
`ALDOR_P_MsgAfterMacEx`	(After Macro Expansion)
`ALDOR_P_MsgExpandedExpr`	Expanded expression was:
`ALDOR_P_MsgSposFileLine`	"%s", line %d:
`ALDOR_P_MsgSposLineChar`	[L%d C%d]
`ALDOR_P_MsgNote`	Note %d

ALDOR_P_MsgSeeNote	(see %s)
ALDOR_P_MsgCfNote	(cf. L%d C%d)
ALDOR_P_MsgCfFarNote	(cf. "%s" L%d C%d)
ALDOR_P_MsgConjunction	and
ALDOR_E_InclBadUnassert	Unassert: property unknown: '%s'.
ALDOR_E_InclIfEof	End of file encountered in '#if'.
ALDOR_E_InclInfinite	Circular include files: %s.
ALDOR_E_InclUnbalElse	Unbalanced '#else'.
ALDOR_E_InclUnbalElseif	Unbalanced '#elseif'.
ALDOR_E_InclUnbalEndif	Unbalanced '#endif'.
ALDOR_E_SysCmdBad	Improper use of '%s' system command.
ALDOR_W_SysCmdUnknown	Unknown system command.
ALDOR_E_ScanBadAftRad	improper number (after radix specification).
ALDOR_E_ScanBadChar	bad character in input.
ALDOR_E_ScanBadExpon	improper number (no digits in exponent).
ALDOR_E_ScanBadRadix	bad radix specification (2 <= radix <= 36).
ALDOR_E_ScanNoDigits	improper number (no digits).
ALDOR_E_ScanOpenString	string not closed.
ALDOR_E_NormMacDecl	Macros must not be given return types.
ALDOR_E_NormMacBadBody	Improper body in 'macro' statement.
ALDOR_W_NormNoId	Couldn't find identifier for documentation
ALDOR_W_NormFornForeign	Foreign(Foreign) is deprecated: use Foreign instead.
ALDOR_W_NormFornBuiltin	Foreign(Builtin) is deprecated: use Builtin instead.
ALDOR_W_NormNullForeign	Foreign() is deprecated: use Foreign instead.
ALDOR_E_MacBadDefn	Improper form for macro definition.
ALDOR_E_MacBadParam	Improper macro parameter (should be an identifier).
ALDOR_E_MacBadParamDecl	Macro parameters must not have type declarations.
ALDOR_E_MacBadArgc	Macro used with incorrect number of arguments.
ALDOR_E_MacBadArg	Macro cannot match the given argument.
ALDOR_E_MacInfinite	Circular macro expansion: %s.
ALDOR_W_MacHides	Definition of macro '%s' hides an outer definition.
ALDOR_W_MacRedefined	Macro '%s' redefined in the same scope.
ALDOR_F_SyntaxOverflow	Parser stack overflow (in state %d).
ALDOR_E_SyntaxError	Syntax error.
ALDOR_E_SyntaxErrorDebug	Syntax error (in state %d).
ALDOR_E_SyntaxErrorHuh	%s.
ALDOR_E_SyntaxNoRecovery	Cannot recover from earlier syntax errors.
ALDOR_E_SyntaxFullError	Syntax error: %s
ALDOR_E_LinUnbalanced	Unbalanced '%s' – missing '%s'.
ALDOR_W_LinUnbalanced	Unbalanced '%s' – missing '%s'.
ALDOR_F_LoadNotAbSyn	This is not an abstract syntax operator.
ALDOR_F_LoadNotFoam	This is not a Foam operator.

ALDOR_F_LoadNotList	Expecting a (parenthesized) list here.
ALDOR_F_LoadNotUnary	Expecting exactly one argument.
ALDOR_F_LoadNotString	Expecting a "quoted" string here.
ALDOR_F_LoadNotSymbol	Expecting a symbol here.
ALDOR_F_LoadNotInteger	Expecting an integer.
ALDOR_F_LoadNotFloat	Expecting a float.
ALDOR_W_CantUnKeywordRef	Unable to consider this ‘ref’ as a non-keyword (compiler limitation: try a simpler definition).
ALDOR_E_ChkBadAssign	Incorrect left-hand side of an assignment.
ALDOR_E_ChkBadDeclare	Improper form appearing within a declaration.
ALDOR_E_ChkBadDefine	Incorrect left-hand side of a definition. Check indentation of succeeding definitions, if any.
ALDOR_W_ChkBadDependent	Bad dependent type detected (or compiler bug)
ALDOR_E_ChkBadFor	Expecting an identifier or single declaration after ‘for’.
ALDOR_E_ChkBadForm	Improper form appearing in ‘%s’ statement.
ALDOR_E_ChkBadGoto	A goto must have a label’s identifier as its target.
ALDOR_E_ChkBadLabel	A label must consist of a single identifier.
ALDOR_E_ChkBadMLambda	Improper macro expansion.
ALDOR_E_ChkBadMacro	Improper macro definition.
ALDOR_E_ChkBadParams	Expecting a comma separated list of parameters.
ALDOR_E_ChkBadParamsDups	Improper duplicate use of parameter name.
ALDOR_E_ChkBadQualification	Improper LHS in $-qualification.
ALDOR_E_ChkBadRecordOrUnion	Duplicate selector/type pair within Record, RawRecord or Union.
ALDOR_E_ChkMissingRetType	Function return type must be specified.
ALDOR_D_ChkUseFromHint	Maybe you want to use ‘import from ...’.
ALDOR_E_ChkSelectSeq	‘select <E> in’ must be followed by a sequence.
ALDOR_E_ChkSelectExits	Unexpected ‘=>’ in select
ALDOR_W_FunnyJuxta	Suspicious juxtaposition. Check for missing ‘;’. Check indentation if you are using ‘#pile’.
ALDOR_W_FunnyColon	Suspicious ‘:’. Do you mean ‘local’ or ‘default’?
ALDOR_W_FunnyEquals	Suspicious ‘=’. Do you mean ‘==’ or ‘:=’ ?
ALDOR_W_FunnyEscape	Escape character ignored. Do you mean ’_’?
ALDOR_W_OldSyntaxAlways	Deprecated syntax: use ‘finally’ instead of ‘always’
ALDOR_W_OldSyntaxCatch	Deprecated syntax: use ‘catch’ instead of ‘but’
ALDOR_W_OldSyntaxThrow	Deprecated syntax: use ‘throw’ instead of ‘except’
ALDOR_W_OldSyntaxUnknown	Deprecated syntax
ALDOR_E_ScoAssAndDef	Cannot both assign and define ‘%s’ in the same scope. Choose ‘==’, ‘:=’, or use as a ‘for’ variable.
ALDOR_E_ScoAssAndRef	Cannot both define and reference ‘%s’ in the same scope. Declare it as a variable instead.

`ALDOR_E_ScoAssTypeId`	'%s' is used in a type, so must be constant, and so cannot be assigned to.
`ALDOR_E_ScoBadLexConst`	A local constant may not have the same name as an outer variable or parameter.
`ALDOR_E_ScoFluidShadow`	A fluid variable cannot shadow an outer non-fluid binding.
`ALDOR_E_ScoBadLoopAss`	Cannot explicitly assign a 'for' variable.
`ALDOR_E_ScoBadParameter`	Improper form appearing in a parameter context.
`ALDOR_E_ScoBadTypeFree`	Free variable '%s' is bound elsewhere with a different type.
`ALDOR_E_ScoDupDefine`	Constant '%s' cannot be redefined.
`ALDOR_E_ScoFreeAndLoc`	Cannot declare '%s' both free and local.
`ALDOR_E_ScoFreeConst`	A constant declared free in an inner scope ('%s') cannot be defined in that scope.
`ALDOR_E_ScoLateFreeLocal`	It is illegal to declare an identifier free or local once it has already been used, defined or assigned.
`ALDOR_E_ScoLibrary`	Cannot assign to or redefine library or archive constant '%s'.
`ALDOR_E_ScoNoFree`	A built-in or foreign function cannot be declared 'free' or 'local'.
`ALDOR_E_ScoNoParm`	A built-in or foreign function cannot have the same signature as a parameter.
`ALDOR_E_ScoNoSet`	A built-in or foreign function cannot be assigned to or defined.
`ALDOR_E_ScoNotBuiltin`	Unknown built-in.
`ALDOR_E_ScoParmLocFree`	Parameters cannot be declared local or free.
`ALDOR_E_ScoSameSig`	A built-in function cannot have the same signature as a foreign function.
`ALDOR_E_ScoParmType`	Parameter type (for %s) must be specified explicitly or with default.
`ALDOR_E_ScoVarOverload`	Variables cannot have different types in the same scope.
`ALDOR_E_ScoUnknownFree`	Cannot find scope in which free variable '%s' is bound.
`ALDOR_W_ScoNotProtocol`	Unknown foreign interface protocol.
`ALDOR_W_ScoBadLocal`	Implicit local '%s' is a parameter, local or explicit free in an outer scope. Add a 'local' declaration if this is what you intended.
`ALDOR_W_ScoFunnyUse`	Identifier '%s' has different declarations in the same scope. Are all implicit and explicit declarations compatible?
`ALDOR_W_ScoBadUse`	Local '%s' is used without being assigned or defined.
`ALDOR_W_ScoLocalNoUse`	Local '%s' is not assigned, defined, or used.
`ALDOR_W_ScoVarDefault`	'%s' has a default type and a different explicit type declaration.
`ALDOR_R_ScoMeaning`	Introducing %s meaning for %s with type %s.

`ALDOR_E_ScoEarlyUse`	Implementation restriction: you cannot use a non-lazy constant outside an 'add' before it has been defined. Perhaps you ought to define '%s' sooner.
`ALDOR_E_StabDupLabels`	Cannot use label '%s' more than once in a given scope.
`ALDOR_R_StabImporting`	Importing %s.
`ALDOR_R_StabImportingQual`	The import was restricted to: %s.
`ALDOR_W_StabNotImporting`	Ignoring explicit import from %s.
`ALDOR_F_LibOutOfDate`	The file '%s.ao' is newer than '%s.ao'.
`ALDOR_F_LibBadVersion`	Library format (obsolete version) in file '%s'. Current library format version %d.%d. Found library format version %d.%d.
`ALDOR_F_LibExportNotFound`	Looking for '%s' with code '%d'. Export not found.
`ALDOR_E_LibBadMagic`	Library format (bad magic number) in file '%s'.
`ALDOR_E_LibBadNumSect`	Library format (bad number of sections) in file '%s'.
`ALDOR_E_LibBadSectHdr`	Library format (bad section header) in file '%s'.
`ALDOR_E_LibBadSectName`	Library format (bad section name) in file '%s'.
`ALDOR_E_LibSectDup`	Library format (duplicate section) in file '%s'.
`ALDOR_E_LibSectLimit`	Library format (too many sections) in file '%s'.
`ALDOR_E_LibSectOffset`	Library format (offset out of range) in file '%s'.
`ALDOR_W_LibRedefined`	Redefinition of library symbol '%s'.
`ALDOR_E_TinNoMeaningForId`	No meaning for identifier '%s'.
`ALDOR_E_TinNoMeaningForLit`	No meaning for %s-style literal '%s'.
`ALDOR_E_TinBadDeclare`	Improper form appearing within a declaration.
`ALDOR_E_TinIfMeans`	The 'if' expression has %d possible types.
`ALDOR_E_TinAssMeans`	Assignment has %d meanings.
`ALDOR_E_TinDefnMeans`	Definition has %d meanings.
`ALDOR_E_TinCantSplitRHS`	This right hand side cannot be split for multiple assignment.
`ALDOR_E_TinAssignCreatesDepType`	The type of this variable includes a variable, '%s'. Consider using '`==`' instead of '`:=`'
`ALDOR_E_TinExprMeans`	Have determined %d possible types for the expression.
`ALDOR_E_TinNMeanings`	There are %d meanings for '%s' in this context.
`ALDOR_E_TinBadGoto`	A goto must have a label's identifier as its target.
`ALDOR_E_TinOpMeans`	Operator has %d possible types.
`ALDOR_E_TinWildExit`	The '=>' is not inside a sequence.
`ALDOR_E_TinWildReturn`	The 'return' is not inside a function.
`ALDOR_E_TinWildYield`	The 'yield' is not inside a 'generate'.
`ALDOR_E_TinContext`	A value is needed but %s does not produce one.
`ALDOR_E_TinContextAssert`	A value is needed but 'assert' does not produce one.
`ALDOR_E_TinContextDo`	A value is needed but 'do' does not produce one.

ALDOR_E_TinContextExit	A value is needed but '=>' does not produce one.
ALDOR_E_TinContextIf	A value is needed but 'if' expression has no 'else'.
ALDOR_E_TinContextRepeat	A value is needed but 'repeat' does not produce one.
ALDOR_E_TinContextSeq	A value is needed but an empty sequence does not produce one.
ALDOR_E_TinCantInferLhs	The type of the assignment cannot be inferred.
ALDOR_E_TinNoGoodOp	There are no suitable meanings for the operator '%s'.
ALDOR_E_TinNoGoodInterp	There is no suitable interpretation for the expression %s
ALDOR_E_TinFirstExitType	The possible type for this %s is %s.
ALDOR_E_TinFirstExitTypes	The possible types for this %s are:
ALDOR_E_TinTypeConstIntro	The interpretation of the type expression
ALDOR_X_TinTypeConstFailed	failed to satisfy the condition that
ALDOR_E_TinCantBeAnalyzed	Cannot determine the meaning of this expression because the type of one of its subexpressions cannot yet be completely analyzed.
ALDOR_E_TinEmbeddedSet	Implicit set within a multi-assign is not yet implemented.
ALDOR_E_TinMultiTry	try expressions must yield a single value (will be fixed later).
ALDOR_E_TinPackedNotSat	Raw record type does not satisfy %s
ALDOR_D_TinNoGoodInterp	There is no suitable interpretation for the expression %s
ALDOR_D_TinNoMeaningForId	No meaning for identifier '%s'.
ALDOR_D_TinSubexprMeans	Subexpression '%s':
ALDOR_D_TinPossTypesLhs	The possible types of the left hand side are:
ALDOR_D_TinPossTypes	The possible types were:
ALDOR_D_TinPossInterps	The possible interpretations of '%s' are:
ALDOR_D_TinPossTypesRhs	The possible types of the right hand side ('%s') are:
ALDOR_D_TinAlternativeMeanings	The following could be suitable if imported:
ALDOR_D_TinOneMeaning	Meaning %d: %s
ALDOR_D_TinContextType	The context requires an expression of type %s.
ALDOR_D_TinMissingExports	The domain is missing some exports.
ALDOR_D_TinMissingExport	Missing %s: %s
ALDOR_D_TinRejectedTypes	These possible types were rejected:
ALDOR_X_TinNoArgumentMatch	rejected because argument %d did not match '%s'.
ALDOR_X_TinParameterMissing	rejected because parameter %d (%s) is missing.
ALDOR_X_TinBadArgumentNumber	rejected because it cannot take %d arguments.
ALDOR_X_TinBadFnType	rejected because the context requires type '%s'.
ALDOR_D_TinRejectedType	The rejected type is %s.
ALDOR_D_TinRejectedTypesForRhs	These possible types for the right hand side were rejected:

ALDOR_D_TinRejectedTypeForRhs	The rejected type for the right hand side is %s.
ALDOR_D_TinShouldUseDoubleEq	You should use %s==%s and not %s.
ALDOR_D_TinAvailableTypesForArg	The available types for argument %d were:
ALDOR_D_TinArgNoMatchParTypes	Argument %d of '%s' did not match any possible parameter type.
ALDOR_D_TinOperatorNoMatch	Operator (argument %d of apply) did not match any possible parameter type.
ALDOR_D_TinMoreMeanings	There are %d meanings for the operator '%s'.
ALDOR_D_TinRetTypesCantContext	No one possible return type satisfies the context type.
ALDOR_D_TinExpectedType	Expected type %s.
ALDOR_D_TinExpectedTypes	Expected one of:
ALDOR_D_TinRejectedRetTypes	These possible return types were rejected:
ALDOR_D_TinOtherDiffArgNum	There are other meanings rejected due to different number of arguments.
ALDOR_D_TinPossSelectorTypes	Possible types for the selector '%s' were:
ALDOR_D_TinPossRetTypeSetBang	Possible return types for 'set!' expression were:
ALDOR_D_TinPossTypesForSetBang	No 'set!' found for any of the possible types for '%s':
ALDOR_D_TinSetBangBadArgNum	There is no 'set!' definition with this number of arguments.
ALDOR_D_TinOneImpMeaning	%s: %s from %s
ALDOR_D_TinOneLexMeaning	%s: %s, a local
ALDOR_D_TinOneLibMeaning	%s: %s, a library
ALDOR_D_TinOneMeaning0	(...): %s
ALDOR_D_TinFirstExitCant	This is not compatible with the types of the other %ss.
ALDOR_N_TinOtherExitType	Here the %s type is %s.
ALDOR_N_TinOtherExitTypes	Here the %s types are:
ALDOR_W_TinNoValReturn	The 'return' gives a value but none is expected.
ALDOR_E_TinReturnNoVal	The 'return' gives no value where one is expected.
ALDOR_R_TinInferring	Inferring %s: %s.
ALDOR_W_TqNotBuiltin	'%s: %s' is not exported by Builtin.
ALDOR_E_GenImpNoRep	Domains with implicit exports must define Rep (as a constant not a macro)
ALDOR_W_GenDomFunNotConst	Function returns a domain that might not be constant (which may cause problems if it is used in a dependent type).
ALDOR_W_GenCatFunNotConst	Function returns a category that might not be constant (which may cause problems if it is used in a dependent type).
ALDOR_W_GenBadDefCycle	Illegal recursive definition: %s
ALDOR_W_GenBadDefOrder	Implementation restriction: the value of '%s' depends on the value of %s. Perhaps you ought to define '%s' later?
ALDOR_M_FintBreakHandler	Execution terminated: use #quit if you want to quit.

ALDOR_M_FintBreakHandler0	Use '#quit' to quit Aldor.
ALDOR_M_FintYesOrNo	Please, answer y or n.
ALDOR_M_FintRedefined	%s redefined.
ALDOR_M_FintOptionState	%s is %s.
ALDOR_M_FintOptionValue	%s is %d.
ALDOR_M_FintUnknownOpt	Unrecognized option. Type: #int %s for help.
ALDOR_M_FintTimings	Comp: %d msec, Interp: %d msec
ALDOR_M_FintGbcStart	Garbage collection...
ALDOR_M_FintGbcEnd	done.
ALDOR_M_FintIntOptionsNoFile	Cannot give files, such as '%s', with '#int options'
ALDOR_M_ShellSyntax	The correct syntax is: #int %s "<shell-command>"
ALDOR_M_CdSyntax	The correct syntax is: #int %s <directory>
ALDOR_M_InvalidDir	Invalid directory.
ALDOR_M_FintOptions	Available options: #int %s [on—off] print the value of an evaluated expression. #int %s [on—off] try to wrap an assignment around the current line. #int %s [on—off] ask for confirmation before redefining something. #int %s [on—off] display timings after every input. #int %s [num] set the limit size of some messages; 0 for no-limit. #int %s ... reset command line options. #int %s perform garbage collection. #int %s "<command>" execute a shell command. #int %s <directory> change current directory. #int %s [0—1—2] display backtrace when an exception occurs. 0: never, 1: only when not caught, 2: always. #int %s display this message. #quit quit the interactive loop.
ALDOR_F_CdFailed	Could not change working directory to '%s'.
ALDOR_F_CcFailed	C compile failed. Command was: %s
ALDOR_F_LinkFailed	Linker failed. Command was: %s
ALDOR_F_BadFType	Cannot handle file '%s' of type '%s'. Try using file type '%s'.
ALDOR_F_WdClobberIn	Output would clobber input file '%s'.

ALDOR_F_WdClobberFile	Output would clobber the source file '%s'.
ALDOR_W_WillObsolete	The file '%s' will now be out of date.
ALDOR_W_RemovingFile	Removing file '%s'.
ALDOR_W_NotCreatingFile	Cannot create file '%s' from input file.
ALDOR_W_NoFiles	No files! Type '%s -help' for help.
ALDOR_F_NoConfig	Could not find aldor.conf
ALDOR_W_CfgError	%s
ALDOR_F_NoFNameProperty	Fortran naming scheme field (%s) is not specified in aldor.conf
ALDOR_F_BadFNameValue	Unrecognised Fortran naming scheme (%s) specified in aldor.conf
ALDOR_F_NoFCmplxProperty	Fortran complex functions field (%s) is not specified in aldor.conf
ALDOR_F_BadFCmplxValue	Unrecognised Fortran complex functions field value (%s) specified in aldor.conf
ALDOR_M_BreakEnter	Aldor compiler break ————————————————————
ALDOR_M_BreakExit	———————————————————————————
ALDOR_M_BreakNoMsg	No message.
ALDOR_M_BreakNoCmd	Unrecognized command: '%s'.
ALDOR_M_BreakMsgPrompt	:::
ALDOR_M_BreakMsgHelpAvail	Help is available.
ALDOR_M_BreakMsgBadNode	Bad node.
ALDOR_M_BreakMsgNoNode	No node.
ALDOR_M_BreakMsgNoStab	No symbol table.
ALDOR_M_BreakMsgNoTypeInfo	No type info yet.
ALDOR_M_BreakMsgAtTop	At top.
ALDOR_M_BreakMsgAtLeaf	At leaf.
ALDOR_M_BreakMsgNoPrev	No prev.
ALDOR_M_BreakMsgNoNext	No next.
ALDOR_M_BreakMsgCantSelect	No such selection.
ALDOR_M_BreakMsgNTypes	The expression has %d possible types.
ALDOR_M_BreakMsg1Type	The expression has the unique type:
ALDOR_M_BreakMsgUsedContext	Used in '%s' context.
ALDOR_M_BreakMsgNotId	Can only ask for meanings of an identifier.

`ALDOR_M_BreakHelp`	Commands are: help – give this help getcomsg – get information on the current message notes – show the notes associated with the current message mselect i – select message i to be the current message mnext – select the next message in the list mprev – select the previous message in the list msg – display the error message again nice – show with pretty printed form ugly – show with more detailed, internal form show – show the current node means – show the possible meanings of the current node use – show how the current node is used seman – show the extra semantic information for the current node scope – show information about the current scope up – use the parent as the current node down – use 0th child as the current node next – use the next sibling as the current node prev – use the previous sibling as the current node home – return to the original node where – returns the line and column location of the current node quit – exit the compiler, showing all messages so far
`ALDOR_H_HelpConfigOpt`	Configuration options: -N file=<file> Specify name of config file. -N sys=<name> Specify system name.

ALDOR_H_HelpCppOpt	C++ generation options: Control the behaviour of '-Fc++'. -P basicfile=<bf> if the filename <bf> (absolute filename) is provided, the standard basic types correspondence between C++ and Aldor will be overridden. If this option is not provided, the compiler uses 'basic.typ' located in $ALDORROOT/include. -P discrim-return Will discriminate functions on the return type by changing the name of the function to 'fnname_return-type'. -P no-discrim-return Won't discriminate functions on the return type. Names of the functions will be as the original, however the code generated for overloaded functions on the return type only won't compile. This option is the default.
ALDOR_H_HelpProductInfo	Contact infodesk@nag.co.uk for product support and information. Use the ALDORbug program for reporting any bugs.
ALDOR_E_SigAbrt	Program fault (abort process).
ALDOR_E_SigBus	Program fault (bus error).
ALDOR_E_SigEmt	Program fault (emulator instruction).
ALDOR_E_SigFpe	Program fault (arithmetic exception).
ALDOR_E_SigHup	User break (hangup).
ALDOR_E_SigIll	Program fault (illegal instruction).
ALDOR_E_SigInt	User break (interrupt).
ALDOR_E_SigPipe	Program fault (write on a pipe with no one to read it).
ALDOR_E_SigDanger	Program fault (paging space low)
ALDOR_E_SigQuit	User break (quit).
ALDOR_E_SigSegv	Program fault (segmentation violation).
ALDOR_E_SigSys	Program fault (bad argument to system call).
ALDOR_E_SigTerm	User break (software termination signal).
ALDOR_E_SigTrap	Program fault (trace trap).
ALDOR_E_SigXcpu	Exceeded time limit imposed by operating system.
ALDOR_E_SigXfsz	Exceeded file size limit imposed by operating system.
ALDOR_E_SigUnknown	Unexpected signal (%d).

`ALDOR_F_SxAlreadyShare`	Share label #nn= previously defined.
`ALDOR_F_SxBadArgumentTo`	Inappropriate argument to function '%s'.
`ALDOR_F_SxBadChar`	Illegal character 0x%x.
`ALDOR_F_SxBadCharName`	Improper character name after # .
`ALDOR_F_SxBadComplexNum`	Improper complex number #C....
`ALDOR_F_SxBadFeatureForm`	Improper feature form following #+ or #-.
`ALDOR_F_SxBadPotNum`	Meaningless potential number '%s'.
`ALDOR_F_SxBadPunct`	Misplaced '%s'.
`ALDOR_F_SxBadToken`	Missing escape in token.
`ALDOR_F_SxBadUninterned`	Package given with '#:'
`ALDOR_F_SxCantMacroArg`	Macro #%c does not take a numeric argument.
`ALDOR_F_SxCantShare`	Share label #nn= not previously defined.
`ALDOR_F_SxInternNeeds`	Intern requires a string.
`ALDOR_F_SxMacroIlleg`	Illegal macro character '#%c'.
`ALDOR_F_SxMacroUndef`	Undefined macro character '#%c'.
`ALDOR_F_SxMacroUnimp`	Unimplemented macro character '#%c'.
`ALDOR_F_SxMustMacroArg`	Macro #n%c requires a numeric argument.
`ALDOR_F_SxNReverseNeeds`	NReverse requires the last cdr of a list to be nil.
`ALDOR_F_SxNumDenNeeds`	%s requires an integer or ratio.
`ALDOR_F_SxPackageExists`	A package with the name %s already exists.
`ALDOR_F_SxReadEOF`	End of file during read.
`ALDOR_F_SxTooManyElts`	Number of elements greater than given size.
`ALDOR_F_StoCantBuild`	Storage allocation error (can't build internal structure).
`ALDOR_F_StoOutOfMemory`	Storage allocation error (out of memory).
`ALDOR_F_StoUsedNonalloc`	Storage allocation error (using non-allocated space).
`ALDOR_F_StoFreeBad`	Storage allocation error (atempt to free unknown space).
`ALDOR_F_CantOpen`	Could not open file '%s'.
`ALDOR_F_CantOpenMode`	Could not open file '%s' with mode '%s'.
`ALDOR_F_CantFindTemp`	Could not find unused temporary file names.
`ALDOR_W_CantUseObject`	Could not use object file '%s'.
`ALDOR_W_CantUseLibrary`	Could not use library file '%s'.
`ALDOR_W_CantUseArchive`	Could not use archive file '%s'.
`ALDOR_W_OverRideLibraryFile`	Current file over-rides existing library in '%s'.
`ALDOR_F_Bug`	Compiler bug: %s.
`ALDOR_W_Bug`	Internal compiler warning: %s
`ALDOR_F_BugExportSymeNotInit`	Compiler bug: I am trying to create a slot for the export '%s:%s'. However, gen0SymeSetInit() has not been used to initialise it so I can not continue (sorry).
`ALDOR_I_PreRelease`	This is a pre-release of %s. 'aldor -h info' for more details.

`ALDOR_I_DemoExpiry`	This is a demo version of %s. 'aldor -h info' for information. This program should not be used after %s
`ALDOR_S_Syme_Label`	label
`ALDOR_S_Syme_Param`	parameter"
`ALDOR_S_Syme_LexVar`	lexical variable
`ALDOR_S_Syme_LexConst`	lexical constant
`ALDOR_S_Syme_Import`	import
`ALDOR_S_Syme_Export`	export
`ALDOR_S_Syme_Extend`	extend
`ALDOR_S_Syme_Library`	library
`ALDOR_S_Syme_Archive`	archive
`ALDOR_S_Syme_Builtin`	builtin
`ALDOR_S_Syme_Foreign`	foreign
`ALDOR_S_Syme_Fluid`	fluid variable
`ALDOR_S_Syme_Trigger`	trigger
`ALDOR_S_Syme_Temp`	temporary

Index