The OL Manual

by

Philip Wadler
Quentin Miller
Martin Raskovsky

Contents

An Introduction to OL

OL is the Oxford Language for functional programming. The OL System includes a text editor, and allows one to create and execute OL programs. This document provides an introduction to OL and the OL system.

1. Scripts and Sessions

Programming in OL consists of two parts. First, one creates a *script* containing a number of declarations. Most of these declarations will be equations defining functions, but one may also write declarations to define new types and new operator symbols. Second, one enters a *session* in which one can type expressions which are evaluated (in the context defined by the script) and the results printed.

For example, one may enter the following definitions in a script:

```
> square x  =  x * x
> cube x    =  x * x * x
```

One then changes from editing the script to editing the session (using the command QUOTE X QUOTE, see later). In the session, one may then type the following:

```
? square (cube 3)
729

(0.05 CPU seconds, 1 page fault, 6 reductions, 10 cells)

?
```

Here, the OL system provided the prompt ?. Following this, the user typed square (cube 3) followed by the "return" key. The OL system responded by displaying the answer, 729, some information about the resources used to calculate this answer, and a new prompt symbol.

2. Function Definitions

Functions are defined equationally, perhaps using recursion and conditionals. For example, we can define a function to raise x to the n'th power as follows:

```
> power x n        =  1                  IF  n = 0
>                     x * power x (n-1)   IF  n > 0
```

One can also define the same function in another way, using pattern matching on the left hand side of the equations:

```
> power' x 0        = 1
> power' x (n+1)    = x * power' x n
```

This definition is equivalent to the previous one, but is shorter and clearer.

In general, the right hand side of an equation may be a conditional form, written using IF and (optionally) OTHERWISE. Also, the right hand side of an equation may be followed by a WHERE clause, which defines the value of one or more variables. For example:

```
> roots a b c  =  [(.~b.+r)./(2.0.*a), (.~b.-r)./(2.0.*a)]  IF  d .> 0.0
>                 [.~b./(2.0.*a)]                           IF  d .= 0.0
>                 []                                        IF  d .< 0.0
>                 WHERE  d = b.^2 .- 4.0.*a.*c
>                        r = sqrt d
```

(For clarity, we have written d .= 0.0 above, but in practice one should write something like near d 0.0, where near x y = abs (x .- y) .< eps.)

Many languages use special symbols (like ;) to delimit the extent of an expression. In OL this information is indicated by indenting. The rule is that the right-hand side of any definition must be indented at least as far as the equal sign. If the right-hand side includes an IF or WHERE clause, each line in the clause must be indented at least as far as the first letter of the keyword IF or WHERE.

3. What's in a name?

Names in OL are of two kinds: alphabetic and symbolic. An alphabetic name begins with a letter, and consists of letters, digits, primes (i.e., single quotes (')), and underbars (_). A symbolic name consists of one or more of the following symbols:

```
+ - * = < > ~ & \ / ^ : # ! . ; | ? @ ` $ % _ { }
```

Underbar is the only symbol that can appear in either kind of name. In alphabetic names, case counts: abc and ABC are distinct names.

Certain names are reserved as keywords:

```
IF  OTHERWISE  WHERE  ELSE  TYPE  DATA  INFIX  INFIXR
=  ::  <-  ->  ><  ..  ;  |
```

These names may not be used as identifiers, but shorter or longer names, such as OTHER or ELSEVIER or <--, are perfectly alright. Sometimes extra parentheses or spaces will be needed to disambiguate. For example, 'a=~b' is read as 'a' followed by '=~' followed by 'b'; to check whether a is equal to ~b we must write 'a=(~b)' or 'a= ~b'.

Every name is either nonfix or infix. For example, square and ~ are nonfix, while + and MOD are infix. By convention, infix names are either symbolic or all upper case, but this convention is not enforced.

4. Basic Data Types

The data types in OL are integers, floating point numbers, booleans, characters, lists, tuples, functions, and user-defined types.

The type of integers is written int. Integers are written in standard decimal notation, such as 42. Unary negation is written ~, as in ~42. The infix operators on integers are: + - * DIV MOD ^.

The type of floating point numbers is written float. Floats are written using a decimal point, as in 6.625. There must be at least one digit on either side of the point, so .5 is incorrect; write 0.5 instead. Unary negation is written .~, as in .~123.4. The infix operators on floats are: .+ .- .* ./ .^. Operations on integers and floats cannot be mixed. Floats are converted to integers by the function entier; integers are converted to floats by the function mkfloat. For example, entier (mkfloat n ./ 2.0) computes, rounding errors aside, the same value as n DIV 2.

The boolean values are written True and False. Boolean values are returned by the comparison operators: = <> < > <= >=. These operators act on any two non-functional values of the same type. For integers and floats, numerical order is used; for characters, the order of the ASCII codes is used; and for lists, tuples, and user-defined types, lexical order is used. The boolean operations are written NOT, AND, and OR.

5. Lists

Lists are written enclosed in square brackets; for example, [1, 2, 3]. The empty list is written [], and a list containing just the number four is written [4]. All of these values have type list int. One may have lists of any type, for example [True, False] has type list bool and [[2.4, 3.0], [], [0.1]] has type list (list float). All elements of a list must have the same type, so [2,True] is illegal.

The operator (:), pronounced 'cons', adds an element to the front of a list. For example, 1 : [2,3] is [1,2,3]. In fact, [1,2,3] is just an abbreviation for 1 : 2 : 3 : [] (which is in turn an abbreviation for 1:(2:(3:[])), since (:) associates to the right.)

Other operations on lists are concatenate (written (++)), length (written (#)), and subscript (written (!)). Also, [m..n] returns the list of integers from m upto n, while [m,n..p] returns the list of integers from m upto p going by steps of (n-m). If m>n, then the sequence will descend to p. Examples of these notations are:

```
[1, 2] ++ [3, 4]  evaluates to   [1, 2, 3, 4]
#[0, 1, 2]        evaluates to   3
[10, 11, 12]!1    evaluates to   11          (Subscripting begins at zero.)
[3 .. 5]          evaluates to   [3, 4, 5]
[5 .. 3]          evaluates to   []
[5, 10 .. 33]     evaluates to   [5, 10, 15, 20, 25, 30]
[5, 0 .. ~33]     evaluates to   [5, 0, ~5, ~10, ~20, ~30]
```

When '..' is followed by '~' or '#', be sure that a space separates the two; otherwise '..~' or '..#' will be read as one name.

One can define new functions on lists using recursion and patterns containing [] and (:). The following function squares every element of a list:

```
> squares []       = []
> squares (x:xs)   = x*x : squares xs
```

For example, (squares [1,2,3]) evaluates to [1,4,9].

6. Characters and Strings

Characters are written enclosed in single quotes, such as 'a'. The function code converts a character to the integer corresponding to its ASCII code, and decode is the inverse. For example, (code 'a') is 97, and (decode 98) is 'b'.

A string is just a list of characters. For example, "abc" is equivalent to ['a','b','c']. One may manipulate strings in the same way as lists; for example, "rat" ++ "her" evaluates to "rather".

Special characters may be written using an escape convention common in programming languages. In particular, a newline is written '\n' and a double quote or backslash character can be included in a string by prefacing it with backslash. For example:

```
"This backslash \\ and\n this quote \" have a newline between them."
```

denotes the sequence of characters

```
This backslash \ and
this quote " have a newline between them.
```

Characters have the type char, so strings have the type list char.

7. Tuples

All elements of a list must have the same type. Collections of objects of different types may be expressed as tuples; for example (1, 2, 'a'), which has type (int >< int >< char). Note that (a >< b >< c), ((a >< b) >< c), and (a >< (b >< c)) are three distinct types: the first is a triple; the latter two are pairs, one component of which is a pair.

The function zip takes a pair of lists and returns a list of pairs. For example, zip ([1,2,3], "abc") evaluates to [(1,'a'), (2,'b'), (3,'c')]. The resulting list has the same length as the shorter of the two argument lists. Thus zip ([1,2,3], "ab") evaluates to [(1,'a'), (2,'b')]. There are similar functions zip3 and zip4, acting on triples and quadruples.

8. Comprehensions

The list comprehension notation makes it easy to describe certain common operations on lists. This notation is very similar to the set comprehension notation from set theory. It will be explained first by example, and then a more precise definition will be given.

The function that squares every element of a list (Section 4) can be defined as follows:

```
> squares' xs  =  [x*x | x <- xs]
```

Similarly, a function that squares only the odd elements of a list can be defined by:

```
> oddsquares xs =  [x*x | x <- xs; x MOD 2 = 1]
```

Vector addition (add corresponding elements of two lists) can be defined using comprehension and the zip function:

```
> vecadd xs ys  =  [x+y | (x,y) <- zip(xs, ys)]
```

The cartesian product of two lists (a list of all pairs with one member from each list) can be defined by:

```
> cp xs ys      =  [(x, y) | x <- xs; y <- ys]
```

For example, (cp [1,2] "abc") returns the list

```
[(1,'a'), (1,'b'), (1,'c'), (2,'a'), (2,'b'), (2,'c')].
```

Note that the second variable changes more rapidly then the first.

The final example is a functional version of Quicksort:

```
> sort []       =  []
> sort (x:xs)   =  sort [u | u<-xs; u<x] ++ [x] ++ sort [u | u <- xs; u >= x]
```

The function sort divides a list into two lists, one consisting of all elements less than x, and the other of all elements greater than or equal to x. The two new lists are then sorted recursively and the results are concatenated together.

Here is a precise, if compact, definition of list comprehensions. A comprehension has the form $[e \mid q_1; \ldots; q_n]$, where e is an expression (of type t) and $q_1, \ldots, q_n$ are qualifiers; the comprehension has type list t. Each qualifier q_i is either a generator or an expression of type boolean. Each generator has the form p_i <- e_i, where p_i is a pattern (of type t_i) and e_i is an expression (of type list t_i). The pattern p_i may contain variables, which are local variables whose scope is the initial expression e and the qualifiers $q_{i+1}, \ldots, q_n$.

9. Lazy Evaluation

Say that we wish to find the first perfect number. (A number is perfect if the sum of its factors, including 1 but not including the number itself, is equal to the number.) If we know that the first perfect number is less than 1000 then we could write:

```
> factors n      =  [i | i <- [1..n-1]; n MOD i = 0]
> perfect n      =  sum (factors n) = n
> first_perfect =  head [n | n <- [1..1000]; perfect n]
```

(The function head returns the first element of a list.) This program is correct, but there are two difficulties. First, the program appears to do rather more work than necessary, since it appears to find a list of all perfect numbers up to 1000, and then throw away all but the first (which is 6). Second, it is annoying to have to give an arbitrary limit, such as 1000.

These problems are solved by the use of an evaluation strategy called *lazy evaluation.* In lazy evaluation, only those parts of an expression necessary to calculate the answer are evaluated. This means that only the first element of the list will be calculated by the program above, so that no extra work will be done. Further, lazy evaluation means that one is allowed to create infinite structures, which are expanded in memory only as needed. For example, [1 ..] returns the infinite list [1, 2, 3, 4, ...], and [1, 3 ..] returns [1, 3, 5, 7, ...].

One can compute the infinite list of all perfect numbers by writing:

```
> perfects  =  [n | n <- [1..]; perfect n]
```

Given this script, the following session can be created:

```
? head perfects
6

(0.54 CPU seconds, 62 page faults, 313 reductions, 947 cells)

? perfects
[6, 28 {Interrupted!}

(8.93 + 1.28 CPU seconds, 166 + 444 page faults, 6170 reductions, 15928 cells)
```

The first term, (head perfects), causes the first perfect number to be printed. The second term, perfects, causes the infinite list of perfect numbers to be printed. The evaluator will continue searching for and printing further elements of the list until an interrupt is typed, as was done here. (The interrupt character is control C.)

Infinite lists may also be created using where clauses. For example the function

```
> dither    =  (yesno, noyes)
>               WHERE    yesno = "YES" : noyes
>                        noyes = "NO"  : yesno
```

returns a pair of infinite lists: (["YES", "NO", "YES", ...], ["NO", "YES", "NO", ...]).

Another property of lazy evaluation is revealed by the following script:

```
> const x y =  x
> loop      =  loop
```

What is the value of (const 42 loop)? In many languages, the answer is that the program enters an infinite loop. But in a language with lazy evaluation, such as OL, the answer is 42. The value of the second argument, loop, is not required to compute the result, so it is not evaluated.

10. Higher-Order Functions

One important property of functional languages is that functions are values, just like integers or lists. Thus, a function may be an argument of a function, the result of a function, an element in a list, and so on. For example, the function

```
> map f xs      =  [f x | x <- xs]
```

takes two arguments, a function f and a list xs, and applies f to every element of xs.

A function may also be *partially applied* by giving only some of its arguments. To illustrate, here is yet another way of defining the function that squares every element of a list:

```
> squares''     =  map square
```

Thus, (map square [1,2,3]) and (squares'' [1,2,3]) both evaluate to [1,4,9].

This style of programming can be quite powerful. For example, the following function defines a common pattern of computation:

```
> foldr f a []      =  a
> foldr f a (x:xs)  =  f x (foldr f a xs)
```

Functions to find the sum and product of all the elements of a list can be defined by:

```
> sum       =  foldr (+) 0
> product   =  foldr (*) 1
```

Notice that to pass an infix operator, such as + or *, as an argument to a function, it must be written as (+) or (*).

11. Variable Declarations

The type-checker automatically deduces the type of all functions used in a program. Thus, type declarations are completely optional. However, programs are often clearer if they contain some type declarations as well.

Variable declarations consist of one or more names followed by a type. For example,

```
> square, cube :: int -> int
> square x  =  x * x
> cube x    =  x * x * x

> squares :: list int -> list int
> squares [ ]     =  [ ]
> squares (x:xs)  =  x^2 : squares xs
```

A type may contain type variables; such types are called *polymorphic*. For example,

```
> map    :: (a -> b) -> list a -> list b
> foldr  :: (a -> b -> b) -> b -> list a -> b
```

Type variables are always single letters, e.g., a or b, while type names are always more than one letter, e.g., char or list; by convention, both are written in lower case. The function map is polymorphic: it has the given type for any instantiation of a and b. For instance, if we let a be int and b be char, then one instance of map is ((int -> char) -> list int -> list char), so that, for instance, the application (map decode) is well-typed, and itself has the type (list int -> list char).

12. Type Synonym Declarations

One may give a new name to an existing type by a type synonym declaration, such as:

```
> TYPE string      =  list char
> TYPE assoc a b   =  list (a >< b)
```

The new names may be used anywhere in place of the equivalent type. For example, [("hi",1), ("lo",0)] has type (assoc string int).

13. Data Type Declarations

The user may define new types. Here is a definition of a tree data type:

```
> DATA tree a  =  Leaf a | Branch (tree a) (tree a)
```

It can be read as follows: 'A tree of a is either a Leaf, which contains an a, or a Branch, which contains a tree of a and a tree of a'. This declaration defines the type tree with *constructors* Leaf and Branch. By convention, constructor names always begin with an upper-case letter. Examples of trees and their types are

```
Branch (Leaf 1) (Leaf 2)           :: tree int
Branch (Leaf 'a') (Leaf 'b')       :: tree char
Branch (Branch (Leaf 1) (Leaf 2))  :: tree (tree int)
```

But Branch (Leaf 1) (Leaf 'b') is not a legal tree, and will cause a type error to be reported.

The constructors Leaf and Branch may appear on the left hand side of equations, as in the following function definition:

```
> reflect (Leaf x)      = Leaf x
> reflect (Branch x y)  = Branch (reflect y) (reflect x)
```

For example, reflect (Branch (Leaf 'o') (Leaf 'h')) returns Branch (Leaf 'h') (Leaf 'o').

New types do not necessarily involve type variables. For example, the Prelude file defines

```
> DATA  bool  =  False | True
```

14. Infix declarations and Sections

We saw earlier how to define a function power such that (power x n) denotes x raised to the n'th power. It may be preferable to define an *infix* name (^) so that (x^n) means the same thing. (In fact, the standard prelude does this.) The necessary declarations are:

```
> INFIXR 8 ^
> x^0       =  1
> x^(n+1)   =  x * x^n
```

The first line says that (^) is an infix operator, of precedence level 8, that associates to the right (R). The precedence level 8 was chosen because it is just higher than multiplication, which is 7, as can be seen by looking at the standard prelude; so (x*x^n) means the same as (x*(x^n)). The definition is identical to that for power, except that terms in the form (power e_1 e_2) are now written in the form (e_1^e_2).

In general, precedence is a number from 0 to 9; higher precedences bind more tightly. An infix name associates to the left if it is declared with INFIX, and to the right if it is declared with INFIXR. If AA and BB are two infixes, of the same precedence and associativity,

then (x AA y BB z) means ((x AA y) BB z) if the operators associate to the left, and (x AA (y BB z)) if the operators associate to the right. By convention, an operator name is either symbolic (like (^)) or consists only of capital letters (like (AA)).

A term of the form x AA y, where AA is infix, is treated the same as (AA) x y. This allows an operator (AA) to be passed as an argument to a higher-order function. (See the examples sum and product, above.) Using an operator as a function by enclosing it in parentheses is called *sectioning*. The following kinds of sections are all permitted:

```
AA x y   = x AA y
(x AA) y = x AA y
(AA y) x = x AA y
```

For example, (./ 2.0) is a function that halves its argument and (1.0 ./) is the reciprocal function.

Sectioning also allows operators to be used on the left hand sides of type declarations. For example,

```
> (^) :: int -> int -> int
```

The names of types and constructors may also be defined as operators. For example, we can replace the previous definition by:

```
> INFIX 4 @
> DATA  tree a  =  'a | (tree a) @ (tree a)
```

Here the symbolic, nonfix name ' is used in place of Leaf, and the symbolic, infix name @ is used in place of Branch. Because OL includes user-defined types and operator symbols, many things which must be 'built in' in other languages can be defined by a normal OL script. The standard prelude gives many examples of this.

15. Ordered Definitions

OL requires that the equations defining a function be *disjoint*, that is, that at most one equation can apply. The following is an *illegal* OL definition of a function to find the last element of a list:

```
> last [x]    =  x
> last (x:xs) =  last xs
```

This is illegal because, for example, (last [1]) can be reduced by either the first equation (giving 1) or by the second equation (giving (last []), an error).

One can write the keyword ELSE before an equation, to indicate that the equation should be tried only after trying to match previously declared equations:

```
> last [x]      = x
> ELSE
> last (x:xs)   = last xs
```

It is a matter of taste whether one prefers to write the above, or to write:

```
> last [x]          = x
> last (x:(x':xs))  = last (x':xs)
```

Here the two equations are disjoint because the second equation only applies to lists with two or more elements.

When a definition contains IF clauses that do not cover all possibilities, then an equation following an ELSE may apply. For example, if we define

```
> apply "add" (x,y) = ("ok", x+y)
> apply "div" (x,y) = ("ok", x DIV y)   IF  y <> 0
> ELSE
> apply name (x,y)  = ("error", 0)
```

then both (apply "div" (3,0)) and (apply "wrong" (4,2)) return ("error",0).

16. Output Formatting

The function show converts any object into a string that represents it. For example,

```
show "hello"     = "\"hello\""
show 3.1416      = "3.1416"
show 'a'         = "\'a\'"
show (1:2:3:[])  = "[1, 2, 3]"
show []          = "[]"
show ""          = "[]"
```

The empty string prints as the empty list since it is impossible to distinguish between the two at run-time (the type information is not available then).

OL can only print objects of type string. If the ? prompt is given an object of another type, show is applied to convert it to a string. Thus, unless x is of type string, both x and (show x) print the same thing:

```
? [1, 6*7, 3]
[1, 42, 3]

? show [1, 6*7, 3]
[1, 42, 3]

? "a\nstring"
a
string

? show "a\nstring"
"a\nstring"
```

Functions useful for control of formatting, such as left or right justification of a string in a field of a given width, are included in the standard prelude.

17. Comments

In most programming languages, comments are indicated by some special symbol. In OL, it is the other way around: everything that is *not* a comment is indicated by beginning the line with the symbol '>'. Also, all comments and program text must be separated by at least one blank line.

This convention allows one to write scripts that read in a natural way. For example, this document is itself a legal OL script! (Note that examples of illegal definitions have a space before the '>', so that they are treated as comments.)

A word of warning: occasionally one will forget to put a '>' at the beginning of a line. Remember to check for this if OL seems to be ignoring part of your script. OL requires that comments and program are separated by a blank line, and this will often catch cases where one omits a '>' by mistake.

18. Errors

OL always reduces an expression as far as possible. If an expression cannot be further reduced because it is in error, OL will mark it as such and perform whatever reductions it can elsewhere. Expressions marked as errors are printed surrounded by curly braces. Here is a sample from a session:

```
? (3 + 4,  map = map,  5 + (2 / (3 - 3)))
(7, {map = map}, {2 / 0})
```

The result is a tuple of three objects, two of which are errors. The first error results from performing a comparison operation on a function, and the second error results from attempting to divide by zero. Note that if the term containing the error is nested, as in (5 + (2 / (3-3))), then only the part of the term that is in error (in this case, {2 / 0}) will be returned.

This treatment of errors is very close to the treatment of ⊥ in domain theory. (The symbol ⊥, pronounced 'bottom', denotes an undefined value, such as the value of an evaluation that enters an infinite loop.) The predefined variable _⊥ can be useful for mimicing the behaviour of ⊥. For example, consider the infinite list ones:

```
> ones          = addone ones
> addone xs     = 1 : xs
```

In domain theory, this list is defined as the limit of the sequence

```
⊥, 1:⊥, 1:(1:⊥), ...
```

Using OL's error mechanism, one can actually compute these approximations to the infinite list of ones:

```
? _⊥
{_⊥}

? addone _⊥
[1] ++ {_⊥}

? addone (addone _⊥)
[1, 1] ++ {_⊥}
```

Of course, one can also print the infinite list of ones directly:

```
? ones
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  {Interrupted!}
```

19. The Editor Interface

OL includes the VED text editor, and users of OL should read the VED manual. All of the normal VED commands may be used to edit at any time during an OL session.

One begins by editing one or more files containing the script, just as one would edit any other files in VED. One may switch to a new file within VED using the E (edit) command; see the VED manual. When the script files are ready, they may be loaded by

the X (execute) command:

QUOTE X $file_1$ ... $file_n$ QUOTE

The script is taken to be the contents of the standard prelude followed by $file_1$ ... $file_n$ in order. To load just the file currently being edited, type QUOTE X - QUOTE.

After the X command, the user will be editing a file called SESSION. Text typed at the end of this file is treated as an OL expression to be evaluated, as shown in the examples of sessions above. Evaluation of an expression in a session (and any long editor command, like FIND) may be interrupted by typing control C.

After some time in the session, the user may wish to edit one of the script files. This may be done, as usual, using the E command. Using the E command without a filename switches one to the last file edited. One may then return to the SESSION file using the X command. The OL system will automatically reload any files of the script that have changed before continuing the session. (Remember, however, to use the X command whenever you wish to change the set of files that is the script; OL does not assume that the script is always the last file or files edited.)

If there is a syntax error in a script file, then when the file is loaded the bell will ring. The file containing the error will be displayed with the cursor at the position where the syntax error was detected. The user may correct the error using the editor, and then resume by giving the X command. If there is more than one error, this process will be repeated. It is usually easy to fix errors, because one is left in the editor with the cursor at the place the error was detected.

Type errors are indicated in a similar way, with the cursor placed at the beginning of the equation that was being analyzed when the type inconsistency was found. Usually, the error will be in this equation, but sometimes the error is in a previous equation and this equation only reveals the inconsistency. One can get additional information about the inconsistency by giving the Y (why) command. This will insert information about the error into the document; the user will usually wish to delete this information after reading it. For convenience, the mark is left at the beginning of the inserted text, which makes it easy to delete with the CUT command (see the VED manual).

20. Using OL On The Sun

To use OL on one of the Sun computers, your .profile must be set up to allow you to use VED. (See the VED manual for details.)

The .profile must also include the following commands:

```
ORWLIB=/usr/lib/Orwell ; export ORWLIB
```

OL may be invoked by typing

```
$ orwell filename
```

This will enter the editor, with file *filename* as the current document. To enter a session directly with the script taken from a (possibly empty) list of files, type

```
$ orwell -x filename
```

The command line may also contain VED flags; see the VED manual.

There are some known problems and deficiencies of the current system. Most important of these are the lack of a proper module system, and the inability to re-direct output to a file other than SESSION. These problems will be fixed in a future version of OL; in the meantime, our apologies.

Users of OL are warned that it is subject to change. Any comments on OL, its implementation, or this document are welcome. If you need help, please see Philip Wadler, Quentin Miller, or Richard Bird.

21. Acknowledgements

Readers familiar with functional programming languages will recognize that OL is mainly a composition of ideas from the work of others, notably Peter Landin's Iswim, Rod Burstall's (and others') NPL and Hope, Robin Milner's (and others') ML and Standard ML, and David Turner's SASL, KRC, and Miranda. In particular, many of the ideas and notations in OL have been taken directly from KRC and Standard ML. (Miranda is a trademark of Research Software Ltd.)

OL has benefitted immensely from discussions with Richard Bird, John Hughes, and Bernard Sufrin. Jeremy Jacob served as OL's first user. Philip Wadler designed the OL language and implemented the interpreter and user interface. Martin Raskovsky implemented the type checker, and Quentin Miller is responsible for current development.

This work was performed while Philip Wadler was supported by a grant from ICL.

Appendix A: The OL Standard Prelude

Infix declarations

```
> INFIXR  1          OR
> INFIXR  2          AND
> INFIX   3          >   >=   =   <>   <=   <   IN
> INFIXR  4          :   ++   --
> INFIX   5          MAX   MIN   .
> INFIX   6          +   -   .+   .-
> INFIX   7          *   DIV   MOD   .*   ./
> INFIXR  8          ^   .^
> INFIX   9          !
```

Type declarations

There are three primitive types, `char`, `int`, and `float`.

```
> DATA bool      = False | True
> DATA list a    = [ ] | a : list a

> TYPE string    = list char

> DATA a >< b              = (a, b)
> DATA a >< b >< c         = (a, b, c)
> DATA a >< b >< c >< d    = (a, b, c, d)
```

Tuples of any size are valid, although only declarations up to quadruples are shown.

Primitive functions

Comparison operations

```
> (=), (<>), (<), (>), (<=), (>=)          :: a -> a -> bool
```

Integer operations

```
> (+), (-), (*), (DIV), (MOD), (^)         :: int -> int -> int
> (~)                                      :: int -> int
> mkfloat                                  :: int -> float
```

Float operations

```
> (.+), (.-), (.*), (./), (.^)              :: float -> float -> float
> (.~), sqrt, exp, log, sin, cos, arctan    :: float -> float
> entier                                    :: float -> int
```

Character operations

```
> code          :: char -> int
> decode        :: int  -> char
```

Strictness

```
> strict        :: (a -> b) -> b -> b
```

(strict f x) forces evaluation of x and then returns (f x).

Display

```
> show          :: a -> string
```

File and keyboard input

```
> filein        :: string -> string
> keyboard      :: bool -> string
```

The function filein takes the name of a file and returns its contents.

A call (keyboard b) returns a list of characters typed at the keyboard; the list may be terminated by type control-Z. If b is True, the characters are echoed as they are typed; if b is False, echoing is not performed. This function is not referentially transparent, since each call may return a different value.

Defined functions

General operations

```
> fix                   :: (a -> a) -> a
> fix f                 = z  WHERE  z = f z

> id                    :: a -> a
> id x                  = x

> const                 :: a -> b -> a
> const x y             = x
```

```
> flip                  :: (a -> b -> c) -> b -> a -> c
> flip f x y            =  f y x

> (.)                   :: (a -> b) -> (c -> a) -> c -> b
> (f . g) x             =  f (g x)

> (MAX)                 :: a -> a -> a
> x MAX y               =  x   IF  x >= y
>                          y   OTHERWISE

> (MIN)                 :: a -> a -> a
> x MIN y               =  x    IF  x <= y
>                          y    OTHERWISE

> fst                   :: a >< b -> a
> fst (x,y)             =  x

> snd                   :: a >< b -> b
> snd (x,y)             =  y
```

Boolean operations

```
> (OR)                  :: bool -> bool -> bool
> True  OR  y           =  True
> False OR  y           =  y

> (AND)                 :: bool -> bool -> bool
> True  AND y           =  y
> False AND y           =  False

> NOT                   :: bool -> bool
> NOT True              =  False
> NOT False             =  True
```

Simple list operations

```
> head                  :: list a -> a
> head (x:xs)           =  x

> tail                  :: list a -> list a
> tail (x:xs)           =  xs

> last                  :: list a -> a
> last xs               =  xs ! (#xs-1)
```

```
> init                  :: list a -> list a
> init xs               =  take (#xs-1) xs

> (++)                  :: list a -> list a -> list a
> [] ++ ys              =  ys
> (x:xs) ++ ys          =  x : (xs ++ ys)

> (#)                   :: list a -> int
> #[]                   =  0
> #(x:xs)               =  1 + #xs

> (!)                   :: list a -> int -> a
> (x:xs) ! 0            =  x
> (x:xs) ! (n+1)        =  xs ! n

> (--)                  :: list a -> list a -> list a
> [] -- ys              =  []
> (x:xs) -- []          =  x:xs
> (x:xs) -- (y:ys)      =  xs -- ys                 IF  x = y
>                          (x : (xs -- [y])) -- ys  OTHERWISE

> (IN)                  :: a -> list a -> bool
> x IN []               =  False
> x IN (x':xs)          =  (x = x') OR (x IN xs)

> sort                  :: list a -> list a
> sort []               =  []
> sort (x:xs)           =  sort [u | u <- xs; u < x]
>                          ++ [x]
>                          ++ sort [u | u <- xs; u >= x]
```

Higher-order operations

```
> map                   :: (a -> b) -> list a -> list b
> map f []              =  []
> map f (x:xs)          =  f x : map f xs

> filter                :: (a -> bool) -> list a -> list a
> filter p []           =  []
> filter p (x:xs)       =  x : filter p xs      IF  p x
>                          filter p xs          OTHERWISE

> repeat                :: (a -> a) -> a -> list a
> repeat f x            =  x : repeat f (f x)
```

```
> foldl                 :: (a -> b -> a) -> a -> list b -> a
> foldl f a []          =  a
> foldl f a (x:xs)      =  strict (foldl f) (f a x) xs

> foldr                 :: (a -> b -> b) -> b -> list a -> b
> foldr f a []          =  a
> foldr f a (x:xs)      =  f x (foldr f a xs)

> foldl1                :: (a -> a -> a) -> list a -> a
> foldl1 f (x:xs)       =  foldl f x xs

> scan                  :: (a -> b -> a) -> a -> list b -> list a
> scan  f a xs          =  a : scan' f a xs
> scan' f a []          =  []
> scan' f a (x:xs)      =  scan f (f a x) xs
```

Fold operations

```
> flatten               :: list (list a) -> list a
> flatten               =  foldr (++) []

> reverse               :: list a -> list a
> reverse               =  foldl (flip (:)) []

> sum                   :: list int -> int
> sum                   =  foldl (+) 0

> product               :: list int -> int
> product               =  foldl (*) 0

> and                   :: list bool -> bool
> and                   =  foldr (AND) True

> or                    :: list bool -> bool
> or                    =  foldr (OR) False

> max                   :: list a -> a
> max                   =  foldl1 (MAX)

> min                   :: list a -> a
> min                   =  foldl1 (MIN)
```

Take and drop operations

```
> take                  :: int -> list a -> list a
> take 0     xs         =  []
> take (n+1) []         =  []
> take (n+1) (x:xs)     =  x : take n xs

> drop                  :: int -> list a -> list a
> drop 0     xs         =  xs
> drop (n+1) []         =  []
> drop (n+1) (x:xs)     =  drop n xs

> takewhile             :: (a -> bool) -> list a -> list a
> takewhile p []        =  []
> takewhile p (x:xs)    =  x : takewhile p xs   IF p x
>                          []                   OTHERWISE

> dropwhile             :: (a -> bool) -> list a -> list a
> dropwhile p []        =  []
> dropwhile p (x:xs)    =  dropwhile p xs       IF p x
>                          x:xs                 OTHERWISE
```

Transposition operations

```
> transpose             :: list a -> list a
> transpose []          =  yss  WHERE  yss = [] : yss
> transpose (xs:xss)    =  [y:ys | (y,ys) <- zip (xs, transpose xss)]

> zip                   :: list a >< list b -> list (a >< b)
> zip ([], ys)          =  []
> zip (x:xs, [])        =  []
> zip (x:xs, y:ys)      =  (x,y) : zip (xs,ys)

> zip3                  :: list a >< list b >< list c -> list (a >< b >< c)
> zip3 ([], ys, zs)     =  []
> zip3 (x:xs, [], zs)   =  []
> zip3 (x:xs, y:ys, [])  =  []
> zip3 (x:xs, y:ys, z:zs)=  (x,y,z) : zip3 (xs,ys,zs)

> zip4                        :: list a >< list b >< list c >< list d
>                                -> list (a >< b >< c >< d)
> zip4 ([], xs, ys, zs)       =  []
> zip4 (w:ws, [], ys, zs)     =  []
> zip4 (w:ws, x:xs, [], zs)   =  []
> zip4 (w:ws, x:xs, y:ys, [])  =  []
> zip4 (w:ws, x:xs, y:ys, z:zs)  =  (w,x,y,z) : zip4 (ws,xs,ys,zs)
```

Layout operations

```
> space          :: int -> string
> space 0        =  []
> space (n+1)    =  ' ' : space n

> laylines       :: list a -> string
> laylines xs    =  [layline i x | (i,x) <- zip ([0..], xs)]
>                   WHERE  layline i x  =  rjustify 4 (show n) ++ ": "
>                                          ++ show x ++ "\n"

> rjustify       :: int -> string -> string
> rjustify n x   =  space (n - #x) ++ x

> ljustify       :: int -> string -> string
> ljustify n x   =  x ++ space (n - #x)
>                   WHERE  s = show x

> cjustify       :: int -> string -> string
> cjustify n x   =  space (m DIV 2) ++ x ++ space (m - m DIV 2)
>                   WHERE  m = 0 MAX (n - #x)
```

Appendix B: OL Grammar

Notation

The notation used to describe the syntax is as follows:

[pattern]	optional	
{pattern}	zero or more repetitions	
(pattern)	grouping	
<pattern>	off-side rule	the text matching the pattern must be indented more than the immediately preceeding token
"LITERAL"	literal	

Syntax

```
program     = decl {decl}
decl        = infixdecl | typedecl | datadecl | vardecl | def

infixdecl   = ("INFIX" | "INFIXR") digit <infix {infix}>
typedecl    = "TYPE" tylhs "=" <tyfunc>
datadecl    = "DATA" tylhs "=" <construct {"|" construct}>
vardecl     = id {"," id} "::" <tyfunc>

construct   = id {typrimary} | tyterm infix tyterm
tyfunc      = tytuple {"->" tytuple}
tytuple     = tyterm {"><" tyterm}
tyterm      = id {typrimary} | tyterm infix tyterm | "(" tyfunc ")"
typrimary   = id | "(" tyfunc ")"
tylhs       = tyvar infix tyvar | id {tyvar}
tyvar       = name

def         = pat "=" rhs {["ELSE"] pat "=" rhs}
rhs         = <(term | conditional) ["WHERE" def {def}]>
conditional = term "IF" <term> {term "IF" <term>} [term "OTHERWISE"]

pat         = pat infix pat | pat "+" integer | id {patprimary}
patprimary  = id | patliteral | pattuple | patlist | "(" pat ")"
pattuple    = "(" pat "," pat {"," pat} ")"
patlist     = "[" [pat {"," pat}] "]"

term        = term infix term | primary {primary}
primary     = id | literal | tuple | listform | section | "(" term ")"
section     = "(" infix primary ")" | "(" primary infix ")"
listform    = list | listcount | listcomp
```

```
tuple       = "(" term "," term {"," term} ")"
list        = "[" [term {"," term}] "]"
listcount   = "[" term ["," term] ".." [term] "]"
listcomp    = "[" term "|" qualifier {";" qualifier} "]"
qualifier   = term | pat "<-" term

patliteral  = integer | character | string
literal     = integer | character | string | float
id          = name | "(" infix ")"
infix       = name
```

Notes on the syntax

A name is infix if it is so declared, and nonfix otherwise. A name may denote either a variable, a constructor, a type, or a type variable. A name appearing in a def or to the left of the "::" in a vardecl denotes a constructor if it is so declared in a DATA declaration, and a variable otherwise. A name appearing in a tyterm or tylhs denotes a type variable if it is one letter long, and a type otherwise.

In a term of the form term infix term it is the precedence of the operators, of course, that determines the abstract syntax tree associated with the term; and similarly for other appearances of infix. Higher precedences bind more tightly.

Lexical grammar

```
integer     = digit {digit}
float       = integer "." integer
character   = single (char | escchar) single
string      = double {char | escchar} double
escchar     = backslash (char | digit [digit [digit]])

name        = alphabetic | symbolic
alphabetic  = letter {letter | digit | single | "_"}
symbolic    = symbol {symbol}
```

Notes on the lexical grammar

In the lexical grammar, single is a single quote or prime (’), double is a double quote ("), and backslash is a backslash (\). A symbol is one of

```
+ - * = < > ~ & \ / ^ : # ! . ; | ? @ ‘ $ % _ { }
```

The other characters that may appear in a program are

```
, " ' ( ) [ ]
```

and letters, digits, and blank space. A blank space is a space, tab, or newline character.

Escape characters in character and string literals are as follows:

\n	newline
\t	tab
\f	formfeed
\r	carriage return
\b	backspace
\\	backslash
\'	single quote
\"	double quote
ddd	character with ASCII code *ddd*, where *ddd* is up to three decimal digits

The same conventions are used in C and Miranda (except in C escape characters of the form *ddd* are in octal, not decimal).

The following are used as key-words or key-symbols:

```
IF  OTHERWISE  WHERE  ELSE  TYPE  DATA  INFIX  INFIXR
=  ::  <-  ->  ><  ..  ;  |
```

Except for =, none of these may be used as names, either infix or nonfix.

Any blank space is ignored, except for in character or string literals, and except for the off-side rule. Each lexeme is as long as possible. Some times extra parentheses or spaces will be needed to disambiguate. For example, 'a=~b' is read as 'a' followed by '=~' followed by 'b'; to check whether a is equal to ~b one must write 'a=(~b)' or 'a= ~b'.

Spaces, tabs, or newlines may not appear inside lexemes, except that spaces or tabs may appear in a character or string literal. Any blank space or newlines between lexemes is ignored (except by the off-side rule).

A comment is any line in which '>' is not the first character on the line. The '>' at the beginning of a non-comment line is ignored. Any comment line adjacent to a non-comment line must contain only blank space (thus, a blank line always seperates programs from comments).