An Introduction to Orwell 6.00

by Philip Wadler
revised by Quentin Miller

Contents

An Introduction to Orwell 6.00

Philip Wadler, 1 April 1985
(revised, December 1985 by Philip Wadler)
(revised, September 1986 by Quentin Miller)
(revised, December 1987 by Quentin Miller)
(revised, October 1988 by Quentin Miller)
(revised, January 1990 by Quentin Miller)

A man may take to drink because he feels himself to be a failure, and then fail all the more completely because he drinks. It is rather the same thing that is happening to the English language. It becomes ugly and inaccurate because our thoughts are foolish, but the slovenliness of our language makes it easier for us to have foolish thoughts.

– George Orwell,
Politics and the English Language, 1946

Orwell is a functional programming language, and it is also a system (including a text editor) for creating and executing programs in that language. This note provides an introduction to the language. At the end are brief instructions on the use of the Orwell system. The Orwell standard prelude is included as an appendix. (Users are warned that the Orwell system is still under development and subject to change.)

The name Orwell was chosen as a reminder that literary values are also important to computer scientists. If we could write programs, documentation and papers that posess a fraction of the power and clarity of the writing of George Orwell, then we would be doing well indeed. Naming a language after Orwell seems particularly appropriate because of his strong concern with the relation between ideas and their expression in words.

1. SCRIPTS AND SESSIONS

Programming in Orwell 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 operator symbols and new types. 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, in a script one may enter the following definitions:

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

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

```
? square (cube 3)
729

(0.02 CPU seconds, 7 reductions, 65 cells)

?
```

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

2. FUNCTION DEFINITIONS

Functions can be defined using recursion and conditionals. For example, one may 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
```

Or, one may 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 exactly equivalent to the previous definition, but it is shorter, and it also makes more clear the inductive nature of the definition of exponentiation.

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*a), (-b-r)/(2*a)],    if d > 0
>                = [-b/(2*a)],                       if d = 0
>                = [],                               otherwise
>                    where d = b*b - 4*a*c
>                          r = sqrt d
```

In many languages, extra symbols (like BEGIN, END and ;) are needed to indicate the extent of an expression. In Orwell, this information is indicated by indenting. The rule is that any expression following an equal sign (in a top–level equation or in a `where` clause) must be indented to appear entirely to the right of the equal sign. Similarly, any expression following an `if` must be indented to appear entirely to the right of the `if`.

A `where` clause in Orwell may contain function definitions of 0 or more arguments. Each of these definitions in turn may contain `if` and `where` clauses. Note that `if` clauses may not themselves contain other `if` or `where` clauses.

3. BASIC DATA TYPES

Life would be dull if we could only write functions on numbers. In addition to numbers, Orwell provides operations on booleans, characters, lists, tuples, and functions, as well as on other types the user may care to define.

Integers are written in standard decimal notation, e.g., `42`. Real numbers are written using a decimal point (there must be at least one digit to the left of the point) or using scientific notation, e.g., `2.0e8`, `3.44e-12`. The `$div` and `$mod` operations can be applied only to integers (giving integral results), while the operations `+ - * / ^` and prefix `-` can be applied to integers or real numbers. The boolean values are written `True` and `False`. Boolean values are returned by the comparison operators: `= ~= < > <= >=`. Boolean values may also be combined using `~ &` and `\/` (*not*, *and*, and *or* respectively).

The name of the number type is `num`, and the name of the boolean type is `bool`.

4. LISTS

Lists are written enclosed in square brackets with commas seperating elements, 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 `[num]`. One may have lists of any type, for example `[True, False]` has type `[bool]` and `[[1, 2.4, 3.0], [], [1.0e-8]]` has type `[[num]]`. All elements of a list must have the same type.

The operator `:` (pronounced *cons*) adds an element to the front of a list. For example, `1 : [2, 3]` is `[1, 2, 3]` and `4 : []` is `[4]`. In fact, `[1, 2, 3]` is just an abbreviation for `1:2:3:[]` (which is in turn an abbreviation for `1:(2:(3:[]))`, since `:` is right associative).

Other operations on lists are append (written `++`), length (written `#`), and subscript (written `!`). Also, `[m..n]` returns the list of numbers from `m` upto `n`, while `[m, n..o]` returns the list of numbers from `m` upto `o` going by steps of `(n-m)`. If `m>n`, then the sequence will decend to 0. For example,

```
[1, 2] ++ [3, 4]    evaluates to [1, 2, 3, 4]
#[0, 1, 2]          evaluates to 3
[10, 11, 12]!1      evaluates to 11
[3..5]              evaluates to [3, 4, 5]
[-8.. -8]           evaluates to [-8]
[5..3]              evaluates to []
[5, 10..33]         evaluates to [5, 10, 15, 20, 25, 30]
[2, 1.5..0]         evaluates to [2, 1.5, 1, 0.5, 0]
```

In the example of `[-8.. -8]`, note that there must be a space between the `..` and the `-` to prevent them from being interpreted as a single symbol.

Note that subscripting begins at zero.

One can define new functions on lists using recursion and patterns on the left-hand side 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]`.

5. 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 97` is `'a'`.

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, `"Hi " ++ "there!"` evaluates to `"Hi there!"`.

Special characters may be written in the same notation as in C. 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:

```
"There's a newline between this backslash \\ and\n this quote \"".
```

Characters have the type `char`, so strings have the type `[char]`.

6. 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')`. The type of `(1, 2, 'a')` is `(num, num, char)`.

The operation `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 result list will have the same length as the shorter of the two input lists. For example, `zip ("hello", [4, 2])` evaluates to `[('h', 4), ('e', 2)]`.

7. 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 (seen previously) can be defined as:

```
> 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 on pairs:

```
> 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:

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

This divides a list into two lists, one of all elements less than some element (say, the first), and the other of all elements greater than or equal to that element. The two new lists are then sorted recursively.

Here is a precise, if compact, definition of list comprehensions. A comprehension has the form `[e | q1; ...; qn]`, where *e* is an expression (of type *t*), and *q1*, ..., *qn* are qualifiers; the comprehension has type `[t]`. Each qualifier *qi* is either a generator or an expression of type boolean. Each generator has the form *pi* `<-` *ei*, where *pi* is a pattern (of type *ti*) and *ei* is an expression (of type `[ti]`). The pattern *pi* may contain variables, which are local variables whose scope is the initial expression *e* and the qualifiers *q(i+1)*, ..., *qn*.

8. 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
> firstperfect    =  hd [n | n <- [1..1000]; perfect n]
```

(The function `hd` 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 the program 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 it in fact does no extra work. Further, it 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, ...]`.

We may return the infinite list of all perfect numbers by writing:

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

Given this script, we can have the following session:

```
? hd perfects
6

(0.12 CPU seconds, 138 reductions, 467 cells)
```

```
? perfects
[6, 28{Interrupted!}

(4.24 CPU seconds, 7976 reductions, 25620 cells)
```

The first term, `hd perfects`, causes the first perfect number to be printed. The second term, perfects, causes the infinite list of perfect numbers to be printed. Orwell will continue searching for the next element of this list forever, or until an interrupt is typed, as was done here. (The default interrupt character is *control–C*.)

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

```
> 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:

```
> k x y           =  x
> loop            =  loop
```

What is the value of `k 42 loop`? In many languages, the answer is that the program enters an infinite loop. But in a language with lazy evaluation, such as Orwell, the answer is `42`.

9. HIGHER–ORDER FUNCTIONS

One important property of functional languages is that functions are values, just like numbers 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`.

Note that `map`, as well as `foldr`, `sum`, and `product` (below) are already defined in the standard prelude.

A function may also be partially "applied" by giving only some of its arguments. For example, 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 `(*)`.

10. TYPE 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.

Type declarations are written in the form `name1, ..., namen :: type`, where there are one or more function names or operators seperated by commas. For example:

```
> square, cube             :: num -> num
> squares                  :: [num] -> [num]
> (+), (*)                 :: num -> num -> num
```

The types of some functions may contain type variables.

```
> map                      :: (a -> b) -> [a] -> [b]
> foldr                    :: (a -> b -> b) -> b -> [a] -> b
```

The type variables must consist of a single lower-case character. If a type contains type variables, it is said to be *polymorphic*. For example, the function `map` has the type given above for any values of `a` and `b`. If we let `a` and `b` both be `num`, then we see that one possible type of `map` is `(num -> num) -> [num] -> [num]`, so that the application `map square` is well-typed, and itself has the type `[num] -> [num]`.

11. TYPE ABBREVIATIONS

One may give a new name to an existing type by a declaration of the form `name == type`. The name may be used anywhere in place of the equivalent type. For example,

```
> string                  == [char]

> addnl                   :: string -> string
> addnl s                 =  s ++ "\n"
```

The above definition of `string` is included in the standard prelude.

12. USER-DEFINED TYPES

The user may define new types. Here is a definition of a tree data type (this example is adapted from a paper by David Turner):

```
> tree x                  ::= Leaf x  |  Pair (tree x) (tree x)
```

It might be read as follows: "A `tree` of `x` is either a `Leaf`, which contains an `x`, or a `Pair`, which contains a `tree` of `x` and a `tree` of `x`." Here `x` is what is called a generic type variable; it stands for the type of elements of the tree, which may be any type. `Pair` and `Leaf` are called constructors; they must be more than one character in length, and they must not begin with a lower-case letter. (Any other symbol must not begin with a capital letter.) For example:

`Pair (Leaf 1) (Leaf 2)`	has type `tree num`
`Pair (Leaf 'a') (Leaf 'b')`	has type `tree char`
`Leaf (Pair (Leaf 1) (Leaf 2))`	has type `tree (tree num)`

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

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

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

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

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

```
> bool             ::= False | True
```

The built-in function `showtype` can be used to find the type of an expression; its use is described below under "OUTPUT FORMATTING".

13. OPERATOR DEFINITIONS

Orwell allows one to define new operators. For example, we can define an operator `**` such that `x**n` denotes `x` raised to the `n`'th integral power.

In order to define a new operator, one must first declare the operator symbol and then define its meaning. One can say that `**` is a new operator symbol that has precedence level 8 and is right associative by writing:

```
> %right 8        **
```

(The precedence level 8 was chosen because it is just higher than the precedence level for multiplication, which is 7, as can be seen in the standard prelude included in the appendix.) The precedence level may be any single digit, and the associativity may be one of `%left`, `%right`, `%non`, or `%prefix`.

If `@` and `` ` `` are two operator symbols of the same precedence and associativity, then `` x @ y ` z `` means `` (x @ y) ` z `` if the operators are left associative, `` x @ (y ` z) `` if the operators are right associative, and is illegal if the operators are non-associative.

The definition of power can now be rewritten:

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

(The reader should compare this with the definition of `power` given earlier.)

In Orwell, a term of the form `x @ y`, where `@` is an infix operator, is treated the same as `(@) x y`. This allows, for instance, operators to be operated on by high-order functions. (See the examples sum and product, above.) Using an operator as a function by enclosing it in parentheses is called sectioning; the following sections are all equivalent to `x @ y`:

```
(@) x y
(x @) y
(@ y) x
```

There is one exception to this. `-` can be used as an infix or a prefix operator, so sections of `-` are interpreted as follows:

`(-)`	binary `-`
`(- a)`	unary `-`
`(a -)`	binary `-`

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

```
> (**)              :: num -> num -> num
```

In addition, any function may be used as an infix operator by prepending it with `$`. `$mod` and `$div` are examples of this. For any function `f`, `x $f y` is equivalent to `f x y`. If `$f` is not explicitly declared in an operator definition, it will be right associative and have the greatest possible precedence level.

The names of types and constructors may also be user defined operators. A constructor operator is declared with `%leftcon`, `%rightcon`, `%noncon`, or `%prefixcon`. For example, the list type is declared in the standard prelude by the following:

```
> %rightcon 1    :

> list a         ::= [] | a : list a
```

(Note that `[a]` is actually an abbreviation for `list a`.) Any constructor may also be used as an operator by prepending it with `$`, as can be done with any function.

Indeed, because Orwell includes user-defined types and operator symbols, many things which must be "built in" in other languages can be defined by a normal Orwell script. The standard prelude gives many examples of this.

14. ORDERED DEFINITIONS

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

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

This is because, e.g., `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.

Note that `last` is defined in the standard prelude.

15. OUTPUT FORMATTING

Orwell can only print objects of type `string`. All other objects must be converted to strings before they can be printed. The primitive function `show` maps any object onto a string. For example,

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

If an expression is typed after the *?* prompt which is not of type `string`, then Orwell will apply `show` to the result before printing it.

There is also a `showtype` primitive function, which maps any object onto a printable representation of its type. For example:

```
showtype "hi"                 evaluates to   "[char]"
showtype (1, "eek!", True)    evaluates to   "(num, [char], bool)"
showtype show                 evaluates to   "a -> string"
showtype (map show)           evaluates to   "[a] -> [string]"
```

16. COMMENTS

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

This convention allows one to write scripts that read in a natural way. For example, this document is itself a legal Orwell script! (Note that examples of illegal definitions or definitions already appearing in the standard prelude 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 Orwell seems to be ignoring part of your script! Orwell requires that comments and program are separated by a blank line, and this will often catch cases where one omits a > by mistake.

17. ERRORS

Orwell always reduces an expression as far as possible. If an expression cannot be further reduced because it is in error, Orwell 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})

(0.12 CPU seconds, 10 reductions, 565 cells)
```

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.) Orwell has a built in function `undefined` which has no defined value. For example, consider the infinite list of ones:

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

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

⊥, addone ⊥, addone (addone ⊥), ...

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

```
? bottom
{undefined}
```

```
? addone bottom
[1] ++ {undefined}

? addone (addone bottom)
[1, 1] ++ {undefined}
```

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!}
```

18. THE VED EDITOR INTERFACE

Orwell can be used with the VED text editor, and users of Orwell with VED should read the VED manual. All of the normal VED commands may be used to edit at any time during an Orwell 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` *file1 ... filen* `QUOTE`

The script is taken to be the contents of the standard prelude followed by *file1*, ... *filen* in order. A `-` may appear in place of any of the *files*; it refers to the file being edited. To just load the file currently being edited, type `QUOTE X - QUOTE`. To just load the prelude, type `QUOTE X . QUOTE`.

After the `X` command, the user will be editing a file called `Session`. Text typed after the `?` prompt at the end of this file is treated as an Orwell 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 `E` command; for convenience, `QUOTE X QUOTE` also returns to the session. The Orwell 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; Orwell does not assume that the script is always the last file edited.)

If there is a syntax error in a script file, then when the file is loaded the bell will ring and the file containing the error will be displayed with the cursor at the position the reader was at when the syntax error was detected. The user may correct the error using the editor, and then resume by giving the QUOTE X QUOTE 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 near the place the error occured.

Type errors are indicated in a similar way, with the cursor 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 some additional information about the inconsistency by giving the Y (why) command. This will insert some 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).

19. THE SCROLLING INTERFACE

Orwell can also be used with a simple teletype–style interface. The interaction between user and interpreter will not be recorded in a "session" file, but the user may use any editor to edit script files.

When Orwell is invoked with this interface, the user will be put into a session right away. In response to the ? prompt given by the interpreter, the user may type any of the following:

COMMAND:	MEANING:
`:x`	re-load current scripts which have changed
`:x .`	just load the prelude
`:x` *file1 ... filen*	load *file1 ... filen*
`:y`	give details of last type error
`:!` *command*	execute *command* under Bourne shell
`:e` *file*	invoke editor containing *file*
`:q`	exit Orwell
`:h`	display commands
`:?`	display commands

Anything else typed in response to a prompt is assumed to be an Orwell expression, which will be reduced and displayed. As in the VED interface, evaluation can be interrupted by typing *control–C*.

The `:e` command invokes the editor named by your shell variable, `EDITOR`. If this variable is not defined, then `ed` is invoked. Note that if you change a current script file, it will not automatically be re-loaded when you exit the editor; you must use the `:x` command. (This is different from the behaviour of the VED interface.)

You can find out what the current files are at any time by giving the `:x` command with no arguments — no files will be read unless they have changed since their last reading, and a list of the current files will be displayed.

If an error is found in a script file which is being loaded, you will be told the line on which the error was found. The actual cause of the error may be on a different line in the same equation. You can use the `:e` command to edit the file in question, then use the `:x` command to try to load the file once you have fixed the error.

20. USING ORWELL ON THE SUN

To use Orwell with the VED interface on a Sun computer, your `.profile` or `.login` must be set up to allow you to use VED. (See the VED manual for details.) Whichever interface you use, you must define the environment variable `ORWELL6LIB` to be `/prg/Orwell6`, and export it. Then your `bin` directory (or some directory in your search path) must contain the following links:

```
orwell -> $ORWELL6LIB/orwell
orwellved -> $ORWELL6LIB/orwellved
```

Orwell with VED may be invoked by typing

`$ orwellved` *filename*

This will enter the editor, with *filename* as the current document. To enter a session directly with *filename* as the current script, type:

`$ orwellved -x` *filename*

When the `-x` flag is used, any number of *filenames* may appear on the command line. If no *filenames* are given, then only the prelude is loaded. The command line may also contain VED flags; see the VED manual.

Another command line option allows you to use a larger or smaller heap than the default 100000 cells. Invoking Orwell with `-n` *number of cells* on the command line allows you to have a heap as small as 10000 or as large as your machine has space for.

To invoke Orwell with the scrolling interface, type:

```
$ orwell file1 ... filen
```

Any number of files may be given (including zero). Orwell will load the prelude and all the files given, and begin a session. The `-n` *number of cells* option can also be used with this command.

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

21. ACKNOWLEDGEMENTS

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

Orwell has benefitted immensely from discussions with Richard Bird, John Hughes, and Bernard Sufrin. Jeremy Jacob served as Orwell's first user. The first implementation of Orwell was written by Phil Wadler and Martin Raskovski. The present implementation was written by Quentin Miller.

This work was performed while the author was supported by a grant from ICL.

At the present time I think we are on the verge of discovering at last what programming languages should really be like. I look forward to seeing many responsible experiments with language design during the next few years; and my dream is that by 1984 we will see a consensus developing for a really good programming language . . .

Will Utopia 84, or perhaps we should call it Newspeak, contain **goto** *statements?*

– Donald Knuth,
Structured Programming with
Go To Statements, 1974

In the year 1984 there was not as yet anyone who used Newspeak as his sole means of communication, either in speech or in writing. The leading articles of The Times were written in it, but this was a tour de force *which could only be carried out by a specialist. It was expected that Newspeak would finally have superseded Oldspeak (or Standard English, as we should call it) by about the year 2050. Meanwhile it gained ground steadily, all Party members tending to use Newspeak words and grammatical constructions more and more in their everyday speech.*

– George Orwell,
Nineteen Eighty–Four, 1949

APPENDIX A: THE ORWELL STANDARD PRELUDE

Orwell Standard Prelude
Phil Wadler, 7 August 1985
(revised Quentin Miller, 13 December 1989)

Operator declarations:

The parser reads `-a` as `$neg a`.

```
> %right      0    ->
> %rightcon   1    :
> %right      1    ++ --
> %right      2    \/
> %right      3    &
> %non        4    > >= = ~= <= < $in
> %right      5    $max $min
> %left       6    + -
> %left       7    * / $div $mod
> %right      8    ^
> %right      9    . !

> %prefix               ~ $neg #
```

Type Declarations:

There are two primitive types, `char` and `num`.

Lists and tuples are built-in; for example,

`[]` is read as `Nil`;
`[x, y, z]` is read as `x:y:z:Nil`;
`[x]` is read as `(list x)` in a type declaration,
`x:Nil` in a value declaration.
`(x,y,z)` is read as `(tuple3 x y z)` in a type declaration,
`(Tuple3 x y z)` in a value declaration.

```
> bool          ::=  False | True
> list a        ::=  Nil | a : list a
> string        ==   [char]
```

Any number of tuple types can be defined using the following pattern; the parser will read them correctly.

```
> tuple2 a b                    ::= Tuple2 a b
> tuple3 a b c                  ::= Tuple3 a b c
> tuple4 a b c d                ::= Tuple4 a b c d
```

Display functions:

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

Primitives:

comparison operations:

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

arithmetic operations:

```
> (+), (-), (*), ($div), ($mod),
> (^), (/)                              :: num -> num -> num

> sqrt, exp, log, sin,
> cos, arctan, entier, ieee             :: num -> num

> ($neg)                                :: num -> num
```

character conversions:

```
> code          ::  char -> num
> decode        ::  num  -> char
```

strictness:

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

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

file input/output:

```
> keyboard      :: bool -> string
> keyboardchar  :: a -> bool
> fileout       :: string -> string -> string
> filein        :: string -> string
```

`keyboard b`	returns the list of characters typed at the keyboard. The input list may be terminated by typing an EOF character (*control-D* by default).

	If `b` is `True`, the characters are echoed as they are typed. If `b` is `False`, echoing is not performed.
`keyboardchar x`	returns `True` if keyboard characters have been typed but not yet read by the `keyboard` primitive; `False` otherwise. (The argument is ignored.)
`fileout f x`	returns the string `x`, marked in a way that causes it to be sent to file `f` when it is printed.
`filein f`	returns the contents of the file `f`.

Warning: `keyboard`, `keyboardchar`, and `filein` are not referentially transparent! For example, `(keyboard b) = (keyboard b)` does not necessarily return `True`; it will return `True` if you type, for example, "abc^D^D".

system io:

```
> system            :: string -> string
```

`system cmd`	returns the standard output and error streams given when `cmd` is evaluated by the operating system.

`system` is, of course, not referentially transparent.

Standard prelude functions: (built-in primitives for speed)

```
> (!)                       :: [a] -> num -> a

* (x:xs) ! 0                =  x
* (x:xs) ! (n+1)            =  xs ! n


> (#)                       :: [a] -> num

* #[]                       =  0
* #(x:xs)                   =  1 + #xs


> (++)                      :: [a] -> [a] -> [a]

* [] ++ ys                  =  ys
* (x:xs) ++ ys              =  x : (xs ++ ys)
```

```
> (--)                  :: [a] -> [a] -> [a]

* xs -- []              =  xs
* xs -- (y:ys)          =  remove xs y -- ys
*                          where remove []     y = []
*                                remove (x:xs) y
*                                  = xs,                 if x = y
*                                  = x : remove xs y,   otherwise


> (.)                   :: (a -> b) -> (c -> a) -> c -> b

* (f . g) x             =  f (g x)


> (&)                   :: bool -> bool -> bool

* True  & y             =  y
* False & y             =  False


> (\/)                  :: bool -> bool -> bool

* True  \/  y           =  True
* False \/  y           =  y


> (~)                   :: bool -> bool

* ~ True                =  False
* ~ False               =  True


> all                   :: (a -> bool) -> [a] -> bool

* all p                 =  and . map p


> and                   :: [bool] -> bool

* and                   =  foldr (&) True


> cjustify              :: num -> string -> string

* cjustify n s          =  space halfm ++ s ++ space (m - halfm)
*                          where m = n - #s
*                                halfm = m $div 2
```

```
> concat                  :: [[a]] -> [a]

* concat                  =  foldr (++) []


> const                   :: a -> b -> a

* const k x               =  k


> copy                    :: num -> a -> [a]

* copy n x                =  take n xs    where xs = x:xs


> drop                    :: num -> [a] -> [a]

* drop 0      xs          =  xs
* drop (n+1) []           =  []
* drop (n+1) (x:xs)       =  drop n xs


> dropwhile               :: (a -> bool) -> [a] -> [a]

* dropwhile p []          =  []
* dropwhile p (x:xs)      =  dropwhile p xs,      if p x
*                         =  x:xs,                otherwise


> filter                  :: (a -> bool) -> [a] -> [a]

* filter p []             =  []
* filter p (x:xs)         =  x : filter p xs,     if p x
*                         =  filter p xs,         otherwise


> foldl                   :: (a -> b -> a) -> a -> [b] -> a

* foldl f a []            =  a
* foldl f a (x:xs)        =  strict (foldl f) (f a x) xs


> foldl1                  :: (a -> a -> a) -> [a] -> a

* foldl1 f (x:xs)         =  foldl f x xs
```

```
> foldr                     :: (a -> b -> b) -> b -> [a] -> b

* foldr f a []              =  a
* foldr f a (x:xs)          =  f x (foldr f a xs)


> foldr1                    :: (a -> a -> a) -> [a] -> a

* foldr1 f [x]              =  x
* foldr1 f (x:y:xs)         =  f x (foldr1 f (y:xs))


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

* fst (x,y)                 =  x


> hd                        :: [a] -> a

* hd (x:xs)                 =  x


> id                        :: a -> a

* id x                      =  x


> ($in)                     :: a -> [a] -> bool

* x $in xs                  =  some (x =) xs


> init                      :: [a] -> [a]

* init [x]                  =  []
* init (x:y:xs)             =  x : init (y:xs)


> iterate                   :: (a -> a) -> a -> [a]

* iterate f x               =  x : iterate f (f x)


> last                      :: [a] -> a

* last [x]                  =  x
* last (x:y:xs)             =  last (y:xs)
```

```
> layn                 :: [string] -> string

* layn xs              =  lay 1 xs
*                         where lay n []      =  []
*                               lay n (x:xs)
*                                 =  rjustify 4 (show n) ++ ") "
*                                    ++ x ++ "\n" ++ lay (n+1) xs


> listmax              :: [a] -> a

* listmax              =  foldl1 ($max)


> listmin              :: [a] -> a

* listmin              =  foldl1 ($min)


> ljustify             :: num -> string -> string

* ljustify n s         =  s ++ space (n - #s)


> map                  :: (a -> b) -> [a] -> [b]

* map f xs             =  [f x | x <- xs]


> ($max)               :: a -> a -> a

* x $max y             =  x,    if x >= y
*                      =  y,    otherwise


> merge                :: [a] -> [a] -> [a]

* merge [] ys          = ys
* merge (x:xs) []      = x:xs
* merge (x:xs) (y:ys)  = x:(merge xs (y:ys)),   if x <= y
*                      = y:(merge (x:xs) ys),   otherwise


> ($min)               :: a -> a -> a

* x $min y             =  x,    if x <= y
*                      =  y,    otherwise
```

```
> or                        :: [bool] -> bool

* or                        =  foldr (\/) False


> product                   :: [num] -> num

* product                   =  foldl (*) 1


> reverse                   :: [a] -> [a]

* reverse                   =  foldl prefix []
*                              where prefix xs x = x:xs


> rjustify                  :: num -> string -> string

* rjustify n s              =  space (n - #s) ++ s


> scan                      :: (a -> b -> a) -> a -> [b] -> [a]

* scan  f a xs              =  a : sc f a xs
*                              where sc f a []     = []
*                                    sc f a (x:xs) = scan f (f a x) xs


> snd                       :: (a,b) -> b

* snd (x,y)                 =  y


> some                      :: (a -> bool) -> [a] -> bool

* some p                    =  or . map p


> sort                      :: [a] -> [a]

* sort                      =  foldr  insert []
*                              where  insert x []     = [x]
*                                     insert x (y:ys)
*                                       = x:y:ys,          if x <= y
*                                       = y:insert x ys,   otherwise


> space                     :: num -> string

* space n                   =  copy n ' '
```

```
> sum                         :: [num] -> num

* sum                         =  foldl (+) 0


> swap                        :: (a -> b -> c) -> b -> a -> c

* swap f x y                  =  f y x


> take                        :: num -> [a] -> [a]

* take 0     xs               =  []
* take (n+1) []               =  []
* take (n+1) (x:xs)           =  x : take n xs


> takewhile                   :: (a -> bool) -> [a] -> [a]

* takewhile p []              =  []
* takewhile p (x:xs)          =  x : takewhile p xs,  if p x
*                             =  [],                  otherwise


> tl                          :: [a] -> [a]

* tl (x:xs)                   =  xs


> until                       :: (a -> bool) -> (a -> a) -> a -> a

* until p f x                 =  x,                   if p x
*                             =  until p f (f x),     otherwise


> zip                         :: ([a], [b]) -> [(a, b)]

* zip ([], ys)                =  []
* zip ((x:xs), [])            =  []
* zip ((x:xs), (y:ys))        =  (x, y) : zip (xs, ys)


> zipwith                     :: (a -> b -> c) -> ([a], [b]) -> [c]

* zipwith f (xs, ys)          =  [f x y | (x, y) <- zip (xs, ys)]
```

APPENDIX B: ORWELL 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 layout must be indented so that every token appears to the right of the token preceding the pattern to which the offside rule applies.
`"literal"`	literal

Syntax

```
program             = decl {decl}
decl                = syndecl | condecl | typedecl | opdecl | def

syndecl             = tylhs "==" <type>
condecl             = tylhs "::=" <construct {"|" construct}>
typedecl            = name {"," name} "::" <type>

name                = var | "(" (var | prefix | infix) ")"

tylhs               = tyvar infix tyvar | prefix tyvar | tylhs1
tylhs1              = tylhsprimary {tyvar}
tylhsprimary        = tyname | "(" (tylhs | tylhssection) ")"
tylhssection        = prefix | infix | infix tyvar | tyvar infix

type                = tyterm1 [infix type]
tyterm1             = prefix tyterm1 | tyterm2
tyterm2             = typrimary | typrimaryname {typrimary}
typrimaryname       = tyname | "(" (type | tysection) ")"
typrimary           = typrimaryname | tyvar | tytuple | tylist
tysection           = prefix | infix | infix tyterm1 | tyterm1 infix
tylist              = "[" type "]"
tytuple             = "(" type "," type {"," type} ")"

construct           = con {typrimary} | typrimary infix typrimary
                    | prefix typrimary

opdecl              = opkind <op {op}>
opkind              = assoc digit | "%prefix" | "%prefixcon"
assoc               = "%left" | "%right" | "%non" |
                      "%leftcon" | "%rightcon" | "%noncon"
```

```
def             = pat "=" rhs {["%else"] pat "=" rhs}
rhs             = (<term> | conditional) [wherepart]
conditional     = ifpart {"=" ifpart"} ["=" otherpart]
ifpart          = <term "," "if" term>
otherpart       = <term "," "otherwise">
wherepart       = "where" <def {def}>

pat             = pat1 [infix pat]
pat1            = prefix pat1 | pat2
pat2            = patprimary | patprimaryname {patprimary}
patprimaryname  = var | "(" (pat | patsection) ")"
patprimary      = patprimaryname | literal | pattuple | patlist
patsection      = prefix | infix | infix pat1 | pat1 infix
pattuple        = "(" pat "," pat {"," pat} ")"
patlist         = "[" [pat {"," pat}] "]"

term            = term1 [infix term]
term1           = prefix term1 | term2
term2           = primary | primaryname {primary}
primaryname     = var | "(" (term | section) ")"
primary         = primaryname | fliteral | tuple | list
section         = prefix | infix | infix term1 | term1 infix
list            = listform | upto | comp
tuple           = "(" term "," term {"," term} ")"
listform        = "[" [term {"," term}] "]"
upto            = "[" term ["," term] ".." [term] "]"
comp            = "[" term "|" [qualifier {";" qualifier}] "]"
qualifier       = term | pat "<-" term

fliteral        = float | literal
literal         = integer | character | string
infix           = op
prefix          = op

tyname          = id
tyvar           = id
con             = id
var             = id
```

Notes On The Syntax

Each prefix or infix `op` must be declared in an `opdecl` before it is used.

An `id` may be either a `tyname`, `tyvar`, `con`, or `var`. If the `id` appears in a `type` or `tylhs` then it is a `tyvar` if it consists of only one character, or a `tyname` if it is longer. Otherwise an `id` is a `con` if it begins with a capital letter, or a `var` if it does not.

Each `op` is either prefix or infix, or both in the unique case of `-`. `(-x)` denotes the negative of `x` rather than a function that subtracts `x` from its argument. Any other ambiguous use of `-` is resolved in favour of the infix (binary) interpretation.

Higher declared precedences bind more tightly. Infix operators beginning with "$", unless they are explicitly declared, bind more tightly than any symbolic operator. Note that prefix operators bind more tightly than infix operators, but less tightly than application.

Lexical Grammar

```
integer           = digit {digit}
float             = integer "." integer ["e" ["-"] integer]
character         = "'" (char | escchar) "'"
string            = """ {char | escchar} """
escchar           = "\" (char | digit [digit [digit]])

pragma            = "%" id

id                = letter {letter | digit | "'" | "_"}
op                = symbol {symbol} | "$" id
```

Notes On The Lexical Grammar

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 or newline character.

Note that tab characters are not recognized by Orwell, either in a script file or in a session. This is because different editors interpret tabs as being equivalent to different numbers of spaces. In order for the off-side rule to work correctly, Orwell must know the exact horizontal position of tokens. Some editors (such as VED) always convert tabs into a number of spaces; so when using such an editor this warning about tabs may be ignored.

Escape characters in character and string literals are as follows :

\n	newline
\t	tab
\f	formfeed
\r	carriage return
\b	backspace
\ddd	character with ASCII code ddd, where ddd is up to three decimal digits

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

Any other character preceeded by a backslash will be read as that literal character, with no special meaning attached.

The following are reserved keywords or key-symbols:

```
if otherwise where = == ::= .. ; | <-
```

Except for `=`, none of these may be used as identifier or operator names. The pragmas and other special symbols that are recognized by the lexical scanner are

```
%left %leftcon %right %rightcon %non %noncon %prefix %prefixcon
%else , ( ) [ ]
```

Pragmas are distinct from identifiers, so for example, `left` is a legal identifier name.

Each lexeme is as long as possible. Sometimes extra spaces or parentheses will be needed to disambiguate. For example, `a=~b` is read as `a` followed by `=~` followed by `b`; to test the equality of `a` and `~b` one must write `a=(~b)` or `a= ~b`.

Blank space may not appear inside lexemes, except that spaces may appear in a character or string literal. Any blank space between lexemes is only significant for 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.