An Introduction to Orwell

(DRAFT)

Philip Wadler

1 April 1985

> 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.)

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
```

In a session, one may then type the following:

```
? square (cube 3)
729

(0.05 seconds, 1 page faults, 6 reductions, 10 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.

FUNCTION DEFINITIONS

The above function definitions are quite simple. More interesting definitions can be made using recursion and guarded equations. 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:

```
> answer x y         =  42          IF  u > 0  AND  v > 0
>                       u - v       OTHERWISE
>                       WHERE  u = a * x
>                              v = a * y
>                              a = (x + y) DIV 2
```

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.

Unlike some languages, WHERE clauses cannot be used to define new functions; these must be defined at the top-level. Also, IF and WHERE clauses may not themselves contain other IF or WHERE clauses.

BASIC DATA TYPES

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

Integers are written in standard decimal notation, e.g., 42. The operations on integers are infix + - * DIV and MOD, and prefix -. (Note that / is not an operator on integers, use DIV instead.) The boolean values are written true and false. Boolean values are returned by the comparison operators: = \= < > <= >=. Boolean values may also be combined using NOT, AND, and OR.

The name of the integer type is Int, and the name of the boolean type is Bool.

LISTS

Lists are written enclosed in 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 `[[1,2,3],[],[1]]` has type `List (List Int)`. 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`. 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]
5..3             evaluates to   []
```

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]`.

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:

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

Characters have the type `Char`, so strings have the type `List Char`.

PAIRS

All elements of a list must have the same type. Collections of objects of different types may be formed by the pair operator, written with a comma, for example `(1, 'a')`. (A pair must

be enclosed in round brackets so that it is not confused with a list.) The type of `(1, 'a')` is `Int >< Char`. (Note: >< is a > and a < typed next to each other.)

The operation `||` takes two lists of the same length and returns a list of pairs; for example `[1,2,3] || "abc"` evaluates to `[(1,'a'), (2,'b'), (3,'c')]`.

There is no "triple" operator, but the , operators associates to the right, so we can write, e.g., `(true,1,'a')` for `(true,(1,'a'))`. The `||` operator also associates to the right, so `"xyz" || [1,2,3] || "abc"` evaluates to `[('x',1,'a'), ('y',2,'b'), ('z',3,'c')]`.

COMPREHENSIONS

The list comprehension notation makes it easy to describe certain common operations on lists. In general, a comprehension has the form:

```
[<expression> | <qualifier>, ..., <qualifier>]
```

where each <qualifier> is either a generator or boolean expression. Each generator has the form:

```
<pattern> <- <expression> .
```

The pattern may contain variables, which are local variables whose scope is delimited by the brackets of the comprehension.

Here are some examples of the use of comprehension. The function that squares every element of a list, seen previously, can also be defined by:

```
> 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 `||` operator on pairs:

```
> vecadd xs ys = [x+y | (x,y) <- 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 Tony Hoare's Quicksort. 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.

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

(A comprehension such as `[u <- xs | u < x]` is an abbreviation for `[u | u <- xs, u < x]`.)

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 <- 1 .. n-1 | n MOD i = 0]
> perfect n     =  sum (factors n) = n
> firstperfect  =  hd [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 <- 1... | perfect n]
```

Given this script, we can have the following session:

```
? hd perfects
6

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

? perfects
[6, 28 (Interrupted!)

(8.93 + 1.28 seconds, 166 + 444 page faults, 6170 reductions, 15928 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 interrupt character is control A.)

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.

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`.

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:

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

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

```
> sum                   =  reduce _+_ 0
> product               =  reduce _*_ 1
```

Notice that to pass an infix operator, such as + or *, as an argument to a function, it must be written as `_+_` or `_*_`. Similarly, prefix and suffix operators, like # or ..., can be passed as arguments by writing `#_` or `_...` .

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):

```
> TYPE  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. Examples of trees and their types are

```
pair (leaf 1) (leaf 2)              :: Tree Int
pair (leaf 'a') (leaf 'b')          :: Tree Char
leaf (pair (leaf 1) (leaf 2))       :: Tree (Tree Int)
```

But `pair (leaf 1) (leaf 'b')` is not a legal tree, and should cause a type error to be reported. (However, the type checker for Orwell is not yet implemented, so it will cause no error at present.)

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')`.

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

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 95 and is right associative by writing:

```
> OP 95 (RIGHT)         ^
```

(The precedence level 95 was chosen because it is just higher than the precedence level for multiplication, which is 90, as can be seen in the standard prelude included in the appendix.)

The definition of power can now be rewritten:

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

The precedence in an operator definition may be any non-negative integer, and the associativity may be one of (LEFT), (RIGHT), or (NON). If @ is an operator, then x @ y @ z means (x @ y) @ z if the operator is left associative, x @ (y @ z) if it is right associative, and is illegal if it is non associative.

The names of types and constructors may also be user defined operators. For example, the pair type is declared in the standard prelude by the following:

```
> OP C (RIGHT)          ,
> OP 60 (RIGHT)         ><
```

```
> TYPE  X >< Y          =  (X , Y)
```

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.

ORDERED DEFINITIONS

Orwell requires that the equations defining a function be disjoint, that is, that at most one equation can apply. The following is <u>not</u> a legal 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 `ORDERED` before a function definition, to indicate that the equations should be tried in order and the first one that matches used:

```
> ORDERED
> last [x]              =  x
> 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.

OUTPUT FORMATTING

Orwell has two output modes. In the first output mode, objects are printed in a form indicating their structure; this is called ? (query) mode. In the second mode, the structure is ignored, and only the components of the structure which are integers or characters are printed; this is called ! (bang) mode. Bang mode is useful when the user wishes to have fine control over the output. The ? or ! prompt printed by the Orwell system may be edited just like any other text, and one switches from one mode to another by simply editing the prompt character.

Here is a transcript of a sample session:

```
? ("The answer\nis ", 6*7)
("The answer\nis " , 42)

! ("The answer\nis ", 6*7)
The answer
is 42
```

```
? 42 = 6*9
false

! 42 = 6*9
```

Note that when `false` is to be printed in bang mode, nothing appears (since `false` is neither an integer nor a character).

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 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.

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,  3 + 'a',  3 + (2 DIV (3 - 3))]
[7, {3 + 'a'}, {3 + {2 DIV 0}}]
```

The result is a list of three objects, two of which are errors. One error results from attempting to add a number and a character, the other from attempting to divide by zero. In general, one may have error objects that are nested, as in `{3 + {2 DIV 0}}`; the innermost curly brackets will contain the initial error, and the outer curly brackets may help in determining the context of the error.

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.) 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

⊥, addone ⊥, addone (addone ⊥), ...

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

list of ones:

```
? bottom
{bottom}

? addone bottom
[1] ++ {bottom}

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

Here bottom prints as an error because it is an undefined name; one could have used any undefined name in its place. 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!}
```

USING ORWELL ON THE VAX

To use Orwell on the computing service VAX2 computer, add the line

```
@DRC4:[PRGPW.ORWELL]ORWELL.COM
```

to your LOGIN.COM file. Orwell may be invoked by typing

```
ORWELL [PRGPW.ORW]DEFS.ORW/X.
```

This will load the standard prelude and begin an Orwell session.

Orwell includes the VED text editor, and users of Orwell should read the VED manual (available in the documentation rack in the terminal room). All of the normal VED commands may be used to edit during an Orwell session. Evaluation of an expression in a session (and any long editor command, like FIND) may be interrupted by typing control A.

One may edit a new script (or any other file) using VED's NEW command. One may load a new script by the command QUOTE X file QUOTE, where file is the name of the file containing the script. One may type just QUOTE X QUOTE to load the file currently being edited. Files loaded are cumulative.

There are some known problems:

Various overflow conditions cause Orwell to role over on its back and die, and a stack dump to be listed on the screen. This will happen, e.g., if the garbage collector does not find any free store.

Old definitions stay in effect until they are replaced; the only way to delete a definition (short of replacing it or ignoring it) is to restart Orwell.

Error messages involving comprehensions are incomprehensible! They involve function names of the form v.XXXXX, where each X is a digit.

File I/O, modules, and type-checking remain to be implemented.

Steps are being taken to fix these problems; in the meantime, my apologies.

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 Phil Wadler (or, if he's unavailable, Bernard Sufrin, Richard Bird, or Jeremy Jacob).

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. Orwell has benefitted immensely from discussions with Richard Bird, John Hughes, and Bernard Sufrin. Jeremy Jacob served as Orwell's first user. Most of the good ideas in Orwell are due to the above people; the bad ideas, I'm afraid, are my own!

> 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: THE ORWELL STANDARD PRELUDE

```
Orwell Standard Prelude
(Philip Wadler, 1 April 1985)


Operator declarations

> OP   0 (RIGHT)        ,
> OP  10 (LEFT)         .
> OP  20 (RIGHT)        OR
> OP  30 (RIGHT)        AND
> OP  40 (RIGHT)        NOT
> OP  50 (NON)          =  <=  >=  <  >  \=
> OP  60 (RIGHT)        :  ++  ||
> OP  60 (LEFT)         --  ><
> OP  70 (NON)          ..  ...
> OP  80 (LEFT)         +  -
> OP  90 (LEFT)         *  MOD  DIV
> OP 100 (NON)          #
> OP 110 (LEFT)         !


Type Declarations

> TYPE    Bool          =  false | true
> TYPE    List T        =  [] | T : List T
> TYPE    A >< B        =  (A , B)

There are two primitive types, Char and Int.
Lists are built-in only in that the parser reads, e.g., [x,y,z] as x:y:z:[].


Primitives

comparison operations:
  _=_ , _<=_ , _<_ , _>_ , _\=_          ::  A -> A -> Bool

arithmetic operations:
  _+_ , _-_ , _*_ , _MOD_ , _DIV_        ::  Int -> Int -> Int
  _-                                     ::  Int -> Int

character conversions:
  code          ::  Char -> Int
  decode        ::  Int  -> Char


The following functions are defined as primitives for speed

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

  map f []              =  []
```

```
map f (x:xs)          =  f x : map f xs

appmap f []           =  []
appmap f (x:xs)       =  f x ++ appmap f xs

appif true x          =  x
appif false x         =  []

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

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

Standard prelude functions

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

> rred f a []         =  a
> rred f a (x:xs)     =  f x (rred f a xs)

> sum                 =  lred _+_ 0
> append              =  rred _++_ []
> rev                 =  lred (C _:_) []

> repeat f x          =  x : repeat f (f x)

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

> [] || []            =  []
> (x:xs) || (y:ys)    =  (x,y) : (xs || ys)

> ORDERED
> tran [xs]           =  [[x] | x <- xs]
> tran (xs:xss)       =  [y:ys | (y,ys) <- xs || tran xss]

> m .. n              =  m : m+1 .. n          IF  m <= n
>                        []                    OTHERWISE

> m ...               =  m : m+1 ...

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

> drop 0 xs           =  xs
> drop (n+1) (x:xs)   =  drop n xs

> last xs             =  xs ! (#xs-1)
```

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

> true  OR  y             =  true
> false OR  y             =  y

> true  AND y             =  y
> false AND y             =  false

> NOT true                =  false
> NOT false               =  true

> hd (x:xs)               =  x
> tl (x:xs)               =  xs

> fst (x,y)               =  x
> snd (x,y)               =  y

> (+ y) x                 =  x + y
```

Note: this is equivalent to either of the following:

```
  +_ y x                  =  _+_ x y
  +_                      =  C _+_
```

```
> (* y) x                 =  x * y
> (MOD y) x               =  x MOD y
> (x :) xs                =  x : xs
> (xs !) n                =  xs ! n
> (! n) xs                =  xs ! n
> (= y) x                 =  x = y
> (\= y) x                =  x \= y
> (> y) x                 =  x > y
> (< y) x                 =  x < y
> (>= y) x                =  x >= y
> (<= y) x                =  x <= y

> K x y                   =  x
> S x y z                 =  x z (y z)
> C f x y                 =  f y x
> (f . g) x               =  f (g x)
```