An Introduction to Obol

Bernard Sufrin

Revision 5 (30th March, 2006)

Abstract

We introduce Obol[1] an imperative Object-oriented variant of ISWIM[2]. The Obol interpreter is written in Java, and Obol and Java objects can easily be used together in the same (Obol, or Java) application.

[1] An *Obol* is (the weight of) half a scruple of silver: Charon's price for the one-way ferry-ride across the river Styx. Obol is not to be confused with `Cobol` "A weak, verbose, and flabby language used by card wallopers to do boring mindless things on dinosaur mainframes." (Jargon: April 2001) "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense." (E.W.Dijkstra). The acronym stands for "COmmon Business-Oriented Language."

[2] ISWIM was a programming language devised by Peter J. Landin and described in his highly influential article, *The next 700 programming languages*, CACM 9(3):157-166 (Mar 1966). The acronym stands for "If you See What I Mean".

Contents

1 Tutorial Introduction **3**

1.1 The Interactive Obol System . . . 3
1.2 Declarations . . . 3
1.3 Exceptions . . . 13
1.4 Sequences and Mappings . . . 14
1.4.1 Sequences . . . 14
1.4.2 Mappings . . . 18
1.5 Records . . . 20
1.6 Data Constructors . . . 22
1.7 Modules . . . 23
1.8 Objects . . . 26

2 Obol Types **30**

2.1 Simple Values . . . 30
2.1.1 ? . . . 30
2.1.2 {} . . . 30
2.1.3 Booleans . . . 30
2.1.4 Numbers . . . 30
2.1.5 Characters . . . 31
2.2 Structured Values . . . 31
2.2.1 Strings . . . 31
2.2.2 Sequences . . . 32
2.2.3 Mappings . . . 33
2.2.4 Records . . . 33

3 Java Objects **34**

3.1 Introduction . . . 34
3.2 Extracts from a Windowing Toolkit . . . 38

A Built-in Modules **41**

B Obol Types **42**

B.1 Built-In Types . . . 42
B.2 Extension Types used in the Java interface . . . 43

C Obol Concrete Syntax **44**

D Gotcha! **48**

D.1 Syntactic 48

D.2 Semantic 49

E Pattern Matching **50**

F The Interactive Obol Window **52**

G Getting and Installing Obol **53**

G.1 Availability 53

G.2 Windows installation 53

G.3 Unix installation 53

1 Tutorial Introduction

1.1 The Interactive Obol System

The interactive Obol system is started with the Unix/Linux command[3]

```
obol
```

The system repeatedly

- Prompts its user with the text >>, then
- reads an Obol phrase followed by a semicolon, then
- evaluates the phrase in the current context, and finally
- prints the resulting value, if any.

The remainder of this tutorial is presented as the transcript of a session with the Obol system in which phrases input by the user are prefixed with >> and system responses are printed below them, without prefix.

Here, for example, we show an interaction in which the Obol interpreter evaluates 30^3:

```
>> 30**3;
27000
```

1.2 Declarations

It is a simple matter to define local procedures and variables. For example, the following expression causes the phrase n^3 to be evaluated in a context in which the name n is associated with a location containing 30.

```
>> let n = 30 in n**3;
27000
```

In the following example, we associate the name x with (a location containing) 10, and the name f with an Obol procedure that computes the value of the factorial function at n, then evaluate the phrase $f\ x$, yielding 3628800.

[3] If you only have the `Obol.jar` file, use `java -jar Obol.jar`, or (Windows) double-click on the Obol.jar file. For more information about the interactive Obol window see Appendix F. For more information about installing Obol see Appendix G.

```
>> let f(n) = if n==0 then 1 else n*f(n-1) fi
   and x    = 10
   in  f(x);
3628800
```

Scope Rules

The scope of the collection of assocations made by such a local declaration is the text of the body of the phrase, as well as the bodies of all functions defined by the same declaration. Furthermore, in a declaration of the form

$$\texttt{let}\ p_1 = e_1\ \texttt{and}\ p_2 = e_2\ \cdots\ \texttt{and}\ p_n = e_n$$

the scope of the association made by $p_i = e_i$ includes $e_{i+1}, e_{i+2}, \cdots e_n$.

```
>> let x=1
   in  let y=x and z=y+1 and x=x+1
       in  [x, y, z];
[2, 1, 2]
```

Nested Scopes

Scopes nest properly, as the following annotated example shows. Overbraces name associations, underbraces show the scope of named assocations.

```
>> let x = 10
   and y = let x = x − 7 and y = x + 1 in x + y
   in  x ∗ y ;
70
```

Persistent Declarations

To declare functions and variables *persistently* in the top-level context we use a special form of `let` phrase – one without an explicit body. In this case the scope of the collection of associations is all subsequent expressions.

In the following example we introduce initialised variables `x` and `y` by associating the name `x` with a location initially containing the value 3, and the name `y` with a location initially containing the value 4. Declarations that appear at the top-level like this are executed for their effect on the context in which subsequent expressions are evaluated. They have no value, so the system prompts for the next phrase immediately.

```
>>  let x = 3 and y = 4;
>>  x;
3
```

```
>>  y;
4

>>  x==4;
true

>>  x==y;
false
```

The value of a variable can be changed with an assignment. Assignments are executed for their effect on variables and, like all other Obol expressions executed for their effects, yield the value {} (pronounced "*null*", or "*nil*").[4]

Assignment

```
>>  x:=42;
{}

>>  x;
42
```

The variable to be assigned to can be determined by an expression, for example:

```
>> if x<y then x else y fi := 99;
>>  (x, y);
(42, 99)
```

Simultaneous assignments are executed by first evaluating all the right-hand sides, then simultaneously updating the locations associated with the variables.

Simultaneous Assignment

```
>>  x, y := y, x;
>>  (x, y);
(99, 42)
```

There is a special case – the *unpacking assignment*: if there is only one expression on the right hand side, and more than one on the left, then the right hand expression must result in a tuple of the right length.

```
>> let rhs = (y, x);
>> x, y := rhs;
>> (x, y);
(42, 99)
```

[4] In a normal interactive session with Obol we can suppress the printing of {} by making the assignment: `system.PrintNil := false;` Henceforth we shall assume this has been done.

There is a corresponding special case for a declaration – the *unpacking declaration*: if there is only one expression on the right hand side, and more than one on the left, then the right hand expression must result in a tuple of the right length.[5] ▹

```
>> let rhs = (y, x);
>> let a, b = rhs;
>> (a, b);
(42, 99)
```

This can be convenient when declaring several variables at once:

```
>> let a, b = (101, 102);
>> (a, b);
(101, 102)
```

BUT NOT:

```
>> let a, b = 101, 102;
Syntax Error: misplaced comma
```

[5] See Note 1 on page i for a more general account of unpacking declarations.

A sequential expression is written in the form $\{e_1; e_2; \cdots e_n\}$ and is evaluated by executing its component expressions one by one. It yields the value of the final expression.

Sequencing

```
>> let x=3 in { x:=x+2; x };
5
```

An iteration always yields null, whatever the value of the expression that is being iterated.

```
>> let x=3 in while x>0 do { x:=x-1; 999 };
{}
```

✂ ✂

The classic "Hello World" program is:

Hello, World!

```
>> println("Hello, World");
Hello, World
```

Here the effect of evaluating the print expression is to output the text `Hello, World`. It then yields the value {} – which is printed by the evaluator only if `system.PrintNil` is true.[6]

[6] In note 2 (page i) we show how to make an independently runnable program.

Many of the remaining Obol control structures will be familiar. In the following example, which demonstrates the use of iteration and choice expressions, we define and use a procedure that computes the greatest common divisor of its arguments. Notice that the body of the procedure is a sequential expression whose last component is the expression denoting the result – Obol has nothing like a `return` command.


```
>> let gcd(m, n) =
   { while m!=n do
     if
      m>n then m:=m-n
     else if
       m<n then n:=n-m
     fi;
     m
   };

>> gcd(36, 15);
3

>> gcd(121, 13);
1
```

✂ ✂

An alternative way of defining the same procedure demonstrates the conditional expression and continued relation notations.[7]

```
>> let gcd(m, n) = 0<m<n => gcd(m, n%m),
                   m>n>0 => gcd(m%n, n),
                   m==0  => n,
                   n==0  => m,
                   m<0   => -gcd(-m, n),
                   n<0   => -gcd(m, -n),
                   n;
```

The conditional expression notation $\mathtt{g} \Rightarrow \mathtt{e}_1, \mathtt{e}_2$ means the same as `if` g `then` $\mathtt{e}_1$ `else` $\mathtt{e}_2$ `fi`. For any relations $\prec^1, \prec^2, \cdots \prec^n$ the "continued relation" notation $\mathtt{e}_1 \prec^1 \mathtt{e}_2 \prec^2 \cdots \prec^{n-1} \mathtt{e}_n$ means the same as $\mathtt{e}_1 \prec^1 \mathtt{e}_2$ && $\mathtt{e}_2 \prec^2 \mathtt{e}_3$ && $\cdots \mathtt{e}_{n-1} \prec^{n-1} \mathtt{e}_n$.

[7] Yes, it's not the most efficient way!

✂ ✂

Type annotations can be associated with any variable or expression, as a way of conveying the programmer's intentions about an invariant property (albeit weak) that is expected to hold of that variable or expression.

A type expression is written in the same superficial syntax as an ordinary expression[8], *but at present type expressions are completely ignored by* Obol *– indeed they don't make it past the parsing stage.*

We expect that type annotations will eventually form the basis for a proper (static) typechecker for Obol.

Examples:

```
>> let factorial:int->int = \(n:int) -> n==0=>1, n*factorial(n-1);
>> let f(x:int) = case factorial x of (n:int)->n;
>> let gcd(m:int, n:int):int =
                  0<m<n => gcd(m, n%m),
                  m>n>0 => gcd(m%n, n),
                  m==0  => n,
                  n==0  => m,
                  m<0   => -gcd(-m, n),
                  n<0   => -gcd(m, -n),
                  n;
```

[8]Except that the symbol -> may be used as an infix

Case expressions make it possible to discriminate on the structure and content of values.


```
>> let greet name =
   case n of "bernard"       -> "What-ho!"
          or "jeff"          -> "G'day!"
          or (first, second) -> "Hello "++first++" "++second
          or _               -> "Greetings, esteemed "++name;
```

A case expression takes the form

```
case expression0 of
     pattern1->expression1
or   ...
or   patternn->expressionn
```

First $expression_0$ is evaluated, yielding value v. Then the patterns are matched[9] against v in order of their occurence. If none matches then the case expression fails – throwing a `bind` exception. Otherwise the expression corresponding to the first pattern that matches is evaluated in the current context, augmented by the result of binding the variables in the pattern to the values at the corresponding positions in v.

The patterns can be arbitrarily complex, and can contain repeated occurences of the same variable. The "variable" _ means "don't care" in a pattern, and never gets bound.

Sequence display patterns match any sequence of the right size, as do tuple displays:

```
>> case [1,2,3] of []->0 or [_]->1 or [_,_]->2 or other->math.maxint
9223372036854775807
>> case [1,2] of []->0 or [_]->1 or [_,_]->2 or other->math.maxint
2
>> case (1,2,3) of ()->0 or (_)->1 or (_,_)->2 or other->math.maxint
1
>> case (1,2) of ()->0 or (_,_)->1 or (_)->2 or other->math.maxint
1
```

A match where there are repeated occurences of the same variable in a pattern will only succeed if the occurences of the variable are all bound to equivalent values.

```
>> case [1,2,3] of [x,x,x] -> true or other -> false;
false
>> case [[1],[1],[1]] of [x,x,x] -> true or other -> false;
true
```

Record patterns match any record containing at least the fields in the pattern.

```
>> case {| a=3; b=4 |} of {| a=x |} -> x;
3
```

[9] See Appendix E for pattern-matching rules.

It can be useful to "hide" variables, so that they cannot inadvertently be referenced or changed. Here is a very simple example, in which the scope of the declaration of `hidden` is the declaration below. That declaration defines two procedures that communicate via the hidden variable: clients of the procedures cannot change the variable because they cannot "see" it.

Hiding variables

```
>> let hidden = 0
   in  let nextNumber() = { hidden:=hidden+1; hidden }
       and reset()      = { hidden:=0 };

>> nextNumber();
1
>> nextNumber();
2

>> hidden;
Error: variable undeclared: hidden

>> reset();
>> nextNumber();
1
```

The formal parameter(s) of a function are actually *patterns*. An attempt to apply a function to an argument that doesn't match its parameter throws an exception.

Parameters are Patterns

```
>> let f(a,b,c) = a+b+c;
>> f[1,2,3];
Error: (a, b, c) does not match [1, 2, 3] at f[1, 2, 3]
>> let f[a,b,c] = a+b+c;
>> f[1,2,3];
6
```

Parameters are passed by value:

```
>> let f x y = x:=y;
   and x=3
   in  { f x 55; x };
3
```

Functions are first-class values, so we can define higher-order functions. For example:

```
>> let compose f g x = f(g x)
   let pair x y = (x, y)
   let both f (x, y) = (f x, f y)
   let par (f, g) x = (f x, g x)
   let add1 x = x+1;
>> par (add1, compose add1 add1) 3;
(4, 5)
>> compose (both (par (add1, compose add1 add1))) (pair 5) 4;
((6, 7), (5, 6))
```

The "lambda expression" written *pat* → *expr* means "that function of *pat* whose value is *expr*."


```
>> both (\ x -> x+1) (3, 4)
(4, 5)

>> (\f -> \g -> \x -> f(g x)) add1 add1 3
5
```

1.3 Exceptions

The evaluation of a `throw` expression results in the throwing of a "catchable" exception – as does the occurence of various other forms of exceptional condition such as division by zero. If such an exception is not caught, then the Obol system reports the dynamic context[10] in which it occured.

```
>> let f(n) =
       if
          n<0 then n
       else if
          n==0 then throw "Non-negative"
       else
          f(n-1)
       fi;
>> f(3);
Error: Non-negative
at throw("Non-negative")
at f(n-1)
at f(n-1)
at f(n-1)
at f(3)
```

The exceptions raised during the execution of an expression can be caught in a (dynamically enclosing) `try` block. When no exception is raised the value of the `try` block is the value of the expression being tried:

```
>> try f(-3) catch s -> { println ("Caught: "++s); 42 };
-3
```

Normally an exception is caught and acted on in the `catch` clause of the closest dynamically-enclosing `try` block.[11] In the following example the exception is not reported, and the result of the try block is the value of the catch expression.

```
>> try f(3) catch s -> { println ("Caught: "++s); 42 };
Caught: Non-negative
42
```

A `then` clause may follow the `catch` clause. In this case the value of the try block is obtained by evaluating the `then` expression – whether or not an exception was thrown during the evaluation of the expression being tried.

[10] *i.e.* the call stack

[11] See Note 3 on page ii for a more detailed account.

```
>> try println(f(3))
       catch s -> { println ("Caught: "++s); 42 };
       then  println "Finished";
Caught: Non-negative
Finished


>> try println(f(-3))
       catch s -> { println ("Caught: "++s); 42 };
       then  println "Finished";
-3
Finished
```

The `catch` clause may be omitted if there is a then clause. In this case the value of the try block is obtained by evaluating the `then` expression after the expression being tried has been evaluated to completion or raised an exception.

```
>> try { print 1; print 2; throw "fail"; print 4 }
       then println 3;
123
```

1.4 Sequences and Mappings

1.4.1 Sequences

The only *predefined* data structures are *finite sequences* and *finite mappings*. Finite sequences behave like functions from an initial subsequence of the natural numbers.[12] A *sequence display* constructs an `immutable` sequence – one whose elements may not be changed.

Sequences

```
>> ["the", "rain", "in", "spain"];
["the", "rain", "in", "spain"]

>> ["the", "rain", "in", "spain"](0);
"the"

>> let con = ["the", "rain", "in", "spain"];
>> #con;
4

>> con(1) := "brain";
Error: ["the", "rain", "in", "spain"] is not a mutable seq.
at con(1):="brain"
```

[12]The precise representation of a sequence depends on how it was constructed, as does the efficiency of the various operations on it. See appendix B for details.

A `for` loop iterates over a sequence:

```
>> for word in con do { print word; print " " };
the rain in spain
```

The elements of mutable sequences (called *arrays*) can be changed. The built-in procedure `newArray` constructs an array of a specific length from a function or int-function-like value.[13] In the following examples we first construct an array from a function that doubles its argument, and then construct one from a sequence.

```
>> let arr = newArray(3, \i -> i*2);
>> arr;
[0, 2, 4]

>> arr(2) := 99; arr;
[0, 2, 99]

>> let var = newArray(#con, con);
>> var(1) := "brain"; var;
["the", "brain", "in", "spain"];
```

An immutable sequence can be constructed using the built-in procedure `newTable`.

```
>> let ro = newTable(3, \i -> i*2);
>> ro;
[0, 2, 4]
>> ro(2) := 99;
Error: [0, 2, 4] is not a mutable seq.
 at ro(2)
 at ro(2):=99
```

✂ ✂

A `range` is a read-only subsequence of the integers, which is represented in a concise form:[14]

Range

```
>> let twenties = 20 # 10;
>> #twenties;
10

>> for i in twenties do print(i, " ");
20 21 22 23 24 25 26 27 28 29

>> for i in 0 # #twenties while i<4 do print(twenties(i), " ");
20 21 22 23
```

[13] *i.e.* any value or object for which the application operator `()` has type consistent with `int->val`.
[14] The infix `#` operator is pronounced "for".

```
>> for i in 0 # #con do println (i, " ", con(i));
0 the
1 rain
2 in
3 spain
```

A range with negative length runs in the opposite direction

```
>> newTable(6, 10#-6);
[10, 9, 8, 7, 6, 5]

>> for i in 10#-6 do print(i, " ");
10  9  8  7  6  5

>> #(10#-6)
6
```

Ranges with matching endpoints behave appropriately when catenated, otherwise they behave just like the sequences they represent:

```
>> (0#5) ++ (5#5);
0#10

>> (10#-5) ++ (5#-5);
10#-10

>> (10#5) ++ (15#5);
10#10

>> (10#5) ++ (15#-5);
[10, 11, 12, 13, 14, 15, 14, 13, 12, 11]
```

✂ ✂

Hitherto we have worked only with simple types, for which the notion of equality is utterly straightforward. When comparing the value of (1 + 4) with the value of (3 + 2) it would be foolish to try to distinguish between these 5s because they came about in different ways – a 5 is a 5 is a 5! There is, therefore, just a single equality on the simple types (such as number), and two elements of a simple type are equivalent if they are equal.

Once we start to work with data structures that can represent more complicated values, we have to take a more refined view, and consider two notions of equality. One is that of equivalence-as-a-value, the other is that of identical-address-in-the-computer. In Obol we use the symbol == to mean "identical" and the symbol === to mean "equivalent".[15]

[15] As a rule of thumb, the *user* of a (composite) data type will mostly be concerned with equivalence, whilst its implementer may also need to be concerned with identity.

Two sequences are equivalent if they contain equivalent elements in the same order.

```
>> [1, 2, 3] === newTable(3, \i->i+1);
true
>> [1, 2, 3] === [1, 2] ++ [3];
true
>> [1, 2, 3] === 1 # 3;
true
>>  [[1], [1,2], [1,2,3]] === [1#1, 1#2, 1#3];
true
```

Equivalent sequences are not necessarily *identical*. On a particular processor the identity of a sequence is a function of its elements in order and the exact moment it was constructed.

```
>> [1, 2, 3] == 1 # 3;
false
>> let s = [1, 2, 3];
>> s == s;
true
>> s == [1, 2] ++ [3];
false
```

Sequences need not necessarily be composed of elements of the same type.[16]


```
>> ["fun", 'f', 23];
["fun", 'f', 23]
```

They can even contain references to themselves! The Obol system prints cyclic data structures specially: printing an ellipsis if it reaches a point in the structure that it has started, but not yet finished, printing.

```
>> let loop = newArray(2, 33) in { loop(1):=loop; loop };
[33, ...]
```

Tools like those familiar from functional programming, such as `zip, map, filter` can easily be built:[17]

[16] To be precise, the Obol interpreter does not require sequences to be homogeneously typed, but some optimising compilers for Obol do have such a requirement.

[17] Indeed they can be built in many slightly different variants, depending on the precise mutabilities required.

```
>> let MAP    f seq = newTable(#seq, \i -> f(seq i))
   and FILTER p seq = cat(newTable(#seq, \i -> p(seq i)=>[seq i],[]))
   and ZIP    seqs  = newTable(#(seqs 1), \i -> newTable(#seqs, \j -> seqs j i));
>> FILTER odd (0#10) where odd n = n%2==1;
[1,3,5,7,9]
>> MAP bit (0#10) where bit n = n%2;
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
>> ZIP [[1,2],[3,4], [5,6,7]];
[(1, 3, 5), (2, 4, 6)]
```

There's also a built-in `filter`: it selects the subsequence of elements of its second argument which satisfy the predicate which is its first argument.

```
>> filter(\i->i%3==0, 0#50);
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

✂ ✂

1.4.2 Mappings

A *finite mapping* display constructs an immutable finite mapping from values to values.

Mapping

```
>> let mapping = [| "foo" -> "a mysterious prefix", "baz"->"a curious suffix" |];
>> mapping("foo");
"a mysterious prefix"
>> for word in mapping do println(word, " is ", mapping word);
foo is a mysterious prefix
baz is a curious suffix
>> (mapping.has("fig"), mapping.has("foo"));
(false, true)
```

Applying a mapping to a key outside its domain causes a `‘subscript` exception to be thrown.

```
>> [|1->2|](3);
Error: key outside domain
at ([| 1->2 |])(3)

>> try [|1->2|](3) catch e->e;
‘subscript ([|1->2|], 3)
```

The catenation operator takes the union of its arguments, except that when there is conflict between its left and right operand it prefers the right.

```
>> mapping ++ [| "foobaz" -> "an odd couplet", "foo" -> "my default identifier" |];
[|"foo"->"my default identifier", "baz"->"a curious suffix",
  "foobaz"->"an odd couplet"|]
```

The `newDict` function constructs a new mutable mapping.

```
>> let d = newDict();
>> d "foo" := "baz";
>> d "baz" := "foo";
>> d;
[|"foo"->"baz", "baz"->"foo"|]
>> d.update([| "foo"->24, "pig"-> 23, "nap"->"won"|]); d;
[|"foo"->24, "pig"->23, "nap"->"won", "baz"->"foo"|]
```

If *m* is a mapping (mutable or not) then *m.keys*() is a sequence containing all the keys (domain elements) of the mapping in an arbitrary order, and *m.values*() is a sequence containing all the values (range elements) of the mapping in an arbitrary order.[18]

It is unwise (though not forbidden) to use mutable structures as keys in mappings.

[18]The more mathematically oriented may prefer the synonyms: *m.dom*() = *m.keys*() and *m.ran*() = *m.values*().

1.5 Records

Records

A record expression takes the form:

$$\{\!|\ \mathtt{id}_1 = \mathtt{expr}_1;\ \cdots \mathtt{id}_n = \mathtt{expr}_n\ |\!\}$$

The expressions are evaluated in turn, and their values associated with the field names $\mathtt{id}_1, \cdots \mathtt{id}_n$.

The field $\mathtt{id}_i$ of a record `r` is selected using the conventional notation $\mathtt{r.id}_i$

For example:

```
>> let ratio = {| num=5; den=6 |};
>> ratio.num;
5
>> ratio.den;
6
```

Individual fields can be assigned to:

```
>> ratio.den:=50;
>> ratio;
{| den = 50; num = 5 |}
```

Record Identity

On a given machine the identity of a record is a function of the exact moment it was created and the values associated with its fields.

```
>> {| num=5; den=6 |} == {| num=5; den=6 |};
false;
>> let rat = {| num=5; den=6 |};
>> rat==rat;
true
>> rat == {| num=5; den=6 |};
false
```

Record Equivalence

By default, two records are equivalent if they have the same field names and the values of corresponding fields are themselves equivalent.

```
>> rat === {| num=5; den=6 |};
true
>> rat === {| num=5; den=6; other={} |};
false
```

Records are frequently used to represent types of value that are somewhat richer in structure than the natural equivalence recognises, so it is useful for us to be able to define specific equivalences for specific types. Obol supports this, and the matter of structural equivalence is taken up at length in Notes 5, and 6 (page iii).

Records are *extended* using the ++ operator.

Record Extension

The result of extending a record r with fieldnames $f_1, \cdots, f_n$ and a record s with fieldnames $g_1, \cdots, g_m$ is a record with fieldnames $f_1, \cdots f_n \cup g_1, \cdots g_m$. Field f of the result has value $s.f$ if f is one of $g_1, \cdots, g_n$, otherwise it has value $r.f$.[19]

For example:

```
>> let rat = {| num=5; den=6 |};
>> let tat = rat ++ {| num=7; kind="slow" |}
>> tat;
{| den=6; kind="slow"; num=7 |}
```

Storage for the fields of the result of an extension is shared with storage for the fields of the operands from which they came.

```
>> tat.num, tat.den := 66, 77;
>> tat;
{| den=77; kind="slow"; num=66 |}
>> rat;
{| den=77; num=5 |}
```

[19] When a field f appears in both operands of a extension, we say that the right hand operand "overrides f".

1.6 Data Constructors

A *data constructor* (or atom) is a (mnemonically-named) constant, written as a backquote followed by a name.

Data Constructors are intended to be used as mnemonic tags in data structures. Those familiar with programming in Haskell, Caml, or Standard ML will recognise data constructors as being akin to the constructors found in those languages.

A data constructor just stands for itself.

```
>> `cons;
`cons
```

A data constructor may also be "applied" to a value – in which case the result is a *construction* – in which the value (known as the *content*) is tagged by the data constructor.

```
>> `cons(1,2);
`cons(1,2)
```

Two constructions are equivalent if they are identical, or if they have the same data constructor as tag, and equivalent content.

```
>> `tag[1,2] === `tag(1#2);
true
```

Data Constructors may appear anywhere in patterns, and a data constructor matches only a data constructor formed from exactly the same name. A pattern of the form *'name*$_1$(*pattern*) matches a value of the form *'name*$_2$(*val*) if, and only if, *name*$_1$ is identical to *name*$_2$ and *pattern* matches *val*.

For example, below we show how to use data constructors in defining a collection of functions to manipulate lists.

```
>> let cons(h, t)     = `cons(h, t)
   and hd(`cons(h,_)) = h
   and tl(`cons(_,t)) = t
   and nil            = `nil
   and null s         = case s of `nil->true or `cons _      -> false
   and len s          = case s of `nil->0    or `cons (_, t) -> 1+len t;
```

1.7 Modules

A module is defined by a module expression. For example, here we define a module that implements a simple random number generator together with various ways of using it.

Defining Modules

```
let random =
module
  let next(bitCount) =
  { seed := (seed * 0x5DEECE66D + 0xB) & ((1 << 48) - 1);
    seed >> (48-bitCount);
  }
  and seed = 253
  and nextByte () = next(8)
  and nextInt  () = next(32)<<32+next(32)
  and nextBool () = next(1)==1
end;
```

The variables defined by a module can be used without importing it.

```
>> newTable(5, \i->random.nextByte());
[239, 37, 127, 20, 29]
```

A module can be imported into a limited scope

Importing Modules

```
>> let randBytes n = import random in newTable(n, \i->nextByte());
>> randBytes(10);
[44, 101, 221, 93, 253, 36, 145, 74, 239, 134]
```

... or into the global context:

```
>> import random;
>> newTable(5, \i->random.nextBool());
[false, true, false, true, false]
```

The global procedure `loadModule` can be used to load Obol code from a file. It wraps the top-level declarations in the file up as a module, and returns that module.[20]

Loading Module Code

For example, suppose that the code to implement the random number generator is present in the file `rand.ob` then loading the file yields a module that has a single binding in it, namely that of `random`:

```
>> loadModule "rand.ob";
module
 random=module
   next=fun next(bitCount)->val
```

[20] If a file has already been loaded this way, then its module value is used again, rather than being reconstructed. To force the reloading of a loaded module it must be unloaded using the library procedure `system.unload`

```
    nextBool=fun nextBool()->val
    nextByte=fun nextByte()->val
    nextInt=fun nextInt()->val
    seed=253
  end
end
```

In order for the module to be any use to the programmer it must be imported.

```
>> import loadModule("rand.ob");
>> newTable(5, \i->random.nextBool());
[false, true, false, false, false]
```

Obol's `import` construct provides a shortcut for importing a module directly from a file – just use a string-valued expression in place of a module-valued expression.

```
>> import "rand.ob";
>> newTable(5, \i->random.nextBool());
[false, true, false, false, false]
```

Modules are first-class values, in the sense that they can be computed as the value of any expression. This makes it easy to define *parameterised modules*.

It would, arguably, be better to parameterise our random number-generating module with an initial value for the seed, and to "hide" the workhorse procedure `next`. The following definition accomplishes this, incidentally adding a generator for reals in the range 0.0-1.0.

```
let random(seed)=
  let next(bitCount) =
  { seed := (seed * 0x5DEECE66D + 0xB) & ((1 << 48) - 1);
    seed >> (48-bitCount);
  }
  and real2E53 = global.math.real(1<<53)
in
module
  let nextByte () = next(8)
  and nextInt  () = next(32)<<32 + next(32)
  and nextBool () = next(1)==1
  and nextReal () = (next(26)<<27+next(27)) / real2E53
end;
```

If this code is placed in the file `random.ob` then we can have several random number generator modules in action at once.

```
>> let r333 = import "random.ob" in random(333)
   and r444 = import "random.ob" in random(444);
```

```
>> for t in 0#5 do println(r333.nextReal(), "\t", r444.nextReal());
0.029830581147327173    0.03977411315859547
0.5833470435760478      0.7639190478719554
0.4555609068093781      0.1525471094440337
0.59304481815736        0.7599605456158061
0.2466127280767071      0.15323355626186452
```

✂ ✂

As a second example, the following defines a sorting utility module parameterised by a module that defines a precedes relation.

```
let sorting(order) =
// Requires    order: module precedes:(val,val)->bool end
module
  let bubble seq =
  // Returns an array that permutes seq with elements ordered
  //         consistently with order.precedes
  { worker (isArray seq=>seq, newArrayOf(seq))
    where worker a =
    { for i in 0#(#a) do
          for j in (i+1)#(#a-i-1) do
              if order.precedes(a j, a i) then a j, a i := a i, a j fi;
      a
    }
  }
  and sort seq =
      // Returns a table that permutes seq with elements ordered
      //         consistently with order.precedes
      #seq<=1 => seq, merge(sort (seq# (#seq/2)),
                            sort (seq<<(#seq/2)))
  and merge(t1, t2) =
  {
     newTable(#t1+#t2, choose)
     where p1 = 0
     and   p2 = 0
     and choose(i) =
         if p1==#t1 then { p2:=p2+1; v } where v=t2 p2 else
         if p2==#t2 then { p1:=p1+1; v } where v=t1 p1 else
         let v1=t1 p1 and v2=t2 p2
         in
             if order.precedes(v1, v2) then
                p1:=p1+1;
                v1
             else
                p2:=p2+1;
                v2
             fi;
         fi
  }
end

and descendingsort = sorting( module precedes(v1, v2) = v1>v2 end )
and ascendingsort  = sorting( module precedes(v1, v2) = v1<v2 end )
```

1.8 Objects

Objects

An object is a record of which one or more fields are procedures.

Procedures defined within records are called `methods`, and within the body of a method, the special symbol `this` denotes the record itself.

Here we define a procedure `rat` which constructs objects that represent rational numbers that can be multiplied. Such objects have a method `mul` that multiplies its own object with another `rat`, and a method `add` that adds another `rat`. They also have an equivalence method and a `_print()` method.

```
>> let rat(n, d) =
   {| num = n;
      den = d;
      mul other = rat(this.num*other.num, this.den*other.den);
      add other = rat(this.num*other.den+this.den*other.num, this.den*other.den);
      _print()  = "%/%"%(this.num, this.den);
      this === other = this.num*other.den == this.den*other.num
   |};

>> rat(6,8);
6/8
```

If an object has a `_print()` method, then that method is invoked by the system whenever it has to generate a printable view of the object – the system generates a printable view of value it returns instead of using the object's default printable view.

If an object has an equivalence method, (defined by `this===other=...`) then its equivalence with other values is decided by that method, rather than by the default structural equivalence for records.

A method is called in the same way as any other procedure; of course it has to be selected from its record first:

```
>> rat(6, 8).mul(rat(1,2));
6/16

>> rat(6, 8).mul(rat(1,2))===rat(3,8);
true
```

Method Calls

When a method, defined by

$$method(arg_1, ...) = body$$

is invoked by

$$object.method(param_1, ...)$$

then the *body* of the method is evaluated, with *this* bound to (the value of) *object*, and $arg_1, ...$ bound to (the values of) $param_1, ...$

Thus, the expression

$$rat(6,8).mul(rat(1,2))$$

is evaluated by evaluating

$$rat(this.num * other.num, this.den * other.den)$$

in a context where $this = rat(6,8)$, and $other = rat(1,2)$. In this context, $this.num = 6$, $this.den = 8$, $other.num = 1$, $other.den = 2$, so the result is a new object constructed by $rat(6,16)$.

---

this Unnecessary

It is not necessary to use *this.* to qualify fieldnames within methods defined in objects: any field f of an object referred to within the body of a method of that object is transformed, at compile-time, into $this.f$. So, for example, the methods *mul* and *add* defined earlier could have been written:

```
mul other = rat(num*other.num, den*other.den);
add other = rat(num*other.den+den*other.num, den*other.den);
```

Inheritance

Our original representation of rationals is somewhat inadequate (no subtraction or division), and we are not entirely happy with the way rationals are printed. We could, of course, simply rewrite the definition, but there is a more convenient way that allows us to exploit our original definition.

We will improve the representation in stages. First we use a *record extend* on the result of `rat` in order to yield a record whose `_print` method presents the number in its simplest form. The new print method overrides the old print method in the result of `ratio`.

```
>>  let ratio(n, d) =
    rat(n, d) ++ {| _print()   = "%/%"%(norm(this.num, this.den));
                    norm(n, d) = (n/g, d/g) where g = gcd(n, d);
                 |};
>>  ratio(8,16);
1/2
```

All well and good? Unfortunately not! The arithmetic methods still yield `rat` results – and they have their own way of printing.

```
>> ratio(8,16).add(ratio(8, 16));
256/256
```

This can be fixed by overriding the `rat` method that is used to construct the results of the arithmetic operations:

```
>>  let ratio(n, d) =
    rat(n, d) ++ {| _print()   = "%/%"%(norm(this.num, this.den));
                    norm(n, d) = (n/g, d/g) where g = gcd(n, d);
                    rat(n, d)  = ratio(n, d);
                 |};

>>  ratio(1,2).add(ratio(8,16));
1/1
```

Of course it would be better to keep our numbers in their simplest form, and the following definition accomplishes this:

```
>>  let ratio(n, d) =
    rat(n, d) ++
    {| _print()   = "%/%"%(this.num, this.den);
       norm(n, d) = (n/g, d/g) where g = gcd(n, d);
       rat(n, d)  = ratio(norm(n, d));
    |};
```

Notice that the tuple `(this.num, this.den)` mentions the names of fields that are not defined within the record where they are used. Although the interpreter will warn us of this, it is not a problem here. Within the body of `_print`, the variable `this` denotes the record from which it was selected: a record equivalent to

```
{| num = ...;
   den = ...;
   mul other      = rat(this.num*other.num, this.den*other.den);
   add other      = rat(this.num*other.den+this.den*other.num,
                        this.den*other.den);
   this === other = this.num*other.den == this.den*other.num
   _print()       = "%/%"%(this.num, this.den);
   norm(n, d)     = (n/g, d/g) where g = gcd(n, d);
   rat(n, d)      = ratio(norm(n, d));
|}
```

It can be very convenient to define infix and prefix operators on types that are represented by records. Here, for example, is a final definement of `rat` that uses infix notation for rational addition and multiplication, and prefix - and / for negation and reciprocation, and then uses negation and reciproaction in the implementation of subtraction and division. The first parameter of infixes (and the only parameter of prefixes) defined within records must be `this`.

```
>> let rational(n, d) =
       ratio(n, d) ++
       {|
          - this         = rational(-num, den);
          / this         = rational(den, num);
          this * other   = mul(other);
          this + other   = add(other);
          this - other   = this + (-other);
          this / other   = this * (/other);
          rat(n, d)      = rational(n, d);
       |};

>> let half=rational(1,2);
>> for n in [half+half, half*half, -half,
             half+(half*half), (half+(half*half))/(half*half)] do println n;
4/4
1/4
-1/2
3/4
3/1
```

It is often necessary to refer to the left-hand operand of an object or record extend from within the right hand operand. This occurs so frequently when constructing objects, that the notation

$$rec_1 \texttt{ with } rec_2$$

may be used as a shorthand for

$$\texttt{let super=}rec_1\texttt{ in super++}rec_2$$

For example

```
>> let rec = {| num=5; den=6 |} with {| previous=super; num=6; den=7 |};
>> rec;
{| den=7; num=6; previous={| den=6; num=5 |} |}
```

2 Obol Types

The details of Obol's predefined types are given in tabular form in appendix B. In this section we give a discursive account of some of the more important types.

2.1 Simple Values

2.1.1 ?

The *completely undefined* Obol value is written `?`. It cannot even be compared with anything, and is useful only for giving "don't care" initial values to variables or the fields of records.

2.1.2 {}

The simplest useful Obol value is {}. Obol phrases that are evaluated solely for their side-effects yield this value.

2.1.3 Booleans

The Boolean values are `true` and `false`. They are equipped with the usual boolean operators. Conjunction and disjunction have the usual "short-cut" meanings.

2.1.4 Numbers

Real numbers are represented by 64-bit floating-point numbers, in the range (`math.minreal`, `math.maxreal`).

They are written, using the usual Scientific notation, in the form

integralpart`.`*fractionalpart*

followed by an optional `E`*exponent*. (For example `3.141593E6` is an approximation to $10^6\pi$.)

Integers are represented as 64-bit signed fixed-point numbers. They are written

- in octal form, as a sequence of decimal digits, starting with 0.
- in hex form, as a sequence of hex digits, prefixed with with 0x.
- in decimal form, as a sequence of decimal digits that does not start with 0.

All numbers are equipped with the usual arithmetic and relational operators, as detailed in Appendix B. Results are integers only if both operands are integers, otherwise they are real.

The built-in `math` module provides a few of the more important real constants and functions

2.1.5 Characters

Characters are encoded in (16-bit) Unicode. They are written as a single ordinary character, or a special character designator, between primes. Special character designators are: `\' \" \r \n \t \f \b \\` or *nnn* where *nnn* is a three-digit octal number.

The characters are equipped with the same relational operators as the numbers, and (in addition)

forward	$c_1 >> n_2$	= the character whose unicode code is n_2 more than c_1
backward	$c_1 << n_2$	= the character whose unicode code is n_2 less than c_1
difference	$c_1 - c_2$	= the difference in unicode codes between c_1 and c_2
Where c_i are characters, and n_i are integers		

For example, the following functions map characters to their unicode encodings, and *vice-versa*.

```
let unichar n = '\000'>>n;
let unicode c = c-'\000';
```

2.2 Structured Values

2.2.1 Strings

Strings are encoded as (16-bit) Unicode sequences. They are written as a sequence of characters (or special character designators) between double-quotes. The length of a string, `s`, is denoted `#s`. The string itself is treated as a function from the indices 0, 1, ... #s-1 to its characters. For example:

```
>> "Hello, World";
"Hello, World"
>> "Hello, World"(0);
'H'
>> "Hello, World"(4);
'o'
```

Strings are catenated with the ++ operator:

```
>> "Ob"++"ol";
"Obol"
```

The following three substring operators are called *take*, *dropleft*, and *dropright*:

```
>> "Obol"#3;
"Obo"
>> "Obol"<<2;
"ol"
>> "Obol">>3;
"O"
```

Strings may be composed using the formatting operator, `%`, as follows:

Formatting

```
>> "% is % for %"%["foo", "on sale", 30.0]
"foo is on sale for 30.0"
```

Successive occurences of % in the formatting string are replaced by the string view of the successive elements of the sequence.

2.2.2 Sequences

A *sequence* is written as a bracketed, comma-separated, sequence of expressions. The length of a sequence, `s`, is denoted `#s`. The sequence itself is treated as a function from the indices 0, 1, ... #a-1.

For example:

```
>> let seq = [1, 2, 3, 4, 5];
>> seq(0);
1
>> seq(1);
2
```

The sequences constructed this way are immutable. Their elements may not be assigned to:

```
>> seq(2):=5;
Error: 3 is not assignable
```

Sequences are catenated with the ++ operator:

```
>>  [0]++seq++[6];
[0, 1, 2, 3, 4, 5, 6]
```

The three subsequence operators are called *take*, *dropleft*, and *dropright*:

```
>> seq#3;
[1, 2, 3]
>> seq<<2;
[3, 4, 5]
>> seq>>3;
[1, 2]
```

The built-in function `newTable` tabulates a function (or function-like value) as a (constant) sequence:[21]

```
>> let facts = newTable(10, f)
>>     where f(n) = n==0=>1, n*f(n-1);
>> facts;
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

The built-in function `newArray` tabulates a function (or function-like value) as a sequence of *locations* that may be assigned to.

```
>> let vfacts = newArray(#facts, facts);
>> vfacts(0):=129;

>> vfacts;
[129, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]

>> let exploded = newArray(#"foobaz", "foobaz");
>> exploded(2), exploded(5) := 'x', 't';

>> string(exploded);
"foxbat";
```

The `newArray` function can accept a simple constant (rather than a function) as its second argument, in which case the constant is replicated the appropriate number of times.

The following built-in functions also construct sequences:

```
newArrayOf seq === newArray(#seq, seq)
newTableOf seq === newTable(#seq, seq)
```

2.2.3 Mappings

See 1.4.2.

2.2.4 Records

See 1.5.

[21] The conditional expression e_0 => e_1, e_2 is equivalent to `if` e_0 `then` e_1 `else` e_3 `fi`

3 Java Objects

3.1 Introduction

In this section, *which should be skipped by people who are not familiar with both Java and OOP terminology*, we outline the interface between Obol and Java.

It is easy to construct an Obol object (a so-called *proxy*) that represents a Java package, interface, or class, and thence to construct a proxy for a Java object of that class, apply a static method of the class, or get hold of a constant defined statically in the class.

In the following example, we acquire a proxy for the package `java.awt`, thence the class, `java.awt.Frame`, then use the class to build (with `.new(...)`) a proxy for a frame.[22]

```
>> let awt     = package("java.awt");
>> let Frame   = awt.Frame;
>> let myFrame = Frame.new("EXAMPLE FRAME");

>> awt;
java.awt;

>> Frame;
java.awt.Frame

>> myFrame;
java.awt.Frame
[frame0,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,
 title=EXAMPLE FRAME,resizable,normal]
```

There is a shortcut if all you want is the frame: the Obol object `java` is a proxy for the "package of packages" whose names begin with `java`, and (by a punning stunt) we are able to interpret Obol dot notation appropriately.

```
>> let myFrame = java.awt.Frame.new("EXAMPLE FRAME");
java.awt.Frame
[frame0,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,
 title=EXAMPLE FRAME,resizable,normal]
```

When calling methods on Java proxies, Obol translates parameters from Obol to Java objects in just the right way. It also translates back from Java values to Obol values when there is an obvious correspondence. Obol proxy objects are made for Java objects, and Obol sequences (of Obol values) for Java arrays.

```
>> java.lang.Double.MAX_VALUE;
1.7976931348623157E308
```

[22] An Obol proxy for a Java object is printed using the string yielded by the `toString()` method of the Java object.

```
>> java.awt.Color.blue;
java.awt.Color[r=0,g=0,b=255]

>> java.math.BigInteger.new("100000000000000000000001");
100000000000000000000001

>> TYPE(java.math.BigInteger.new("100000000000000000000001"));
‘javaobject "java.math.BigInteger"

>> java.math.BigInteger.new("100000000000000000000001").bitLength();
80
```

Perhaps the most useful part of all this,[23] is that we can extend Java objects with Obol methods, and we can build Obol implementations of Java interfaces.

In the following example we define a function that takes an Obol procedure `f`, and returns an Obol proxy for a (Java) implementation of the Java interface `java.awt.event.ActionListener`. The Java implementation invokes the Obol method `actionPerformed` when its own `actionPerformed` method is called, and this invokes `f` (with the name of the button's command).

```
let actionFunction(f) =
    java.awt.event.ActionListener.new
    {|
       actionPerformed(event) = f (event.getActionCommand())
    |};
```

The `new` method of (the Obol proxy for) a Java interface expects a record whose methods have names identical to those specified by the Java interface, and yields an Obol proxy for a (Java) implementation of the Java interface.

In the Java `awt` a button is associated with an action by adding to the button an `ActionListener` whose `actionPerformed` method implements the action.

Below we take a slightly different approach by defining an Obol procedure `Button`. It builds an Obol button object that can be used as an `awt` component. The button object has a `withFunction` method that arranges for its argument to be called with an appropriate string argument whenever the button is pressed.

```
let Button(text) =
java.awt.Button.new(text) with
{| withFunction(f) =
   { super.addActionListener(actionFunction(f));
     this
   }
|};
```

[23] and certainly the most challenging part to implement!

The result of `Button` is an object composed of (a proxy for) a `java.awt.Button` extended with an Obol class that implements `withFunction`, by calling the `addActionListener` method in the kernel (super) object, then returns the complete extended object.

We can use these tools to build a (very simple) user interface that simply reports the "press" events on each of four buttons.

```
let pressed command = println(command) in
let frame = java.awt.Frame.new("Application") in
{  frame.setLayout(java.awt.GridLayout.new(2,2));
   for name in ["Mumble", "Grumble", "Rhubarb", "Crumble"] do
       frame.add(Button(name).withFunction(pressed));
   frame.pack();
   frame.show()
}
```

Later in this section we give a partial example of how one might build a convenient interface to the Java Abstract Windowing Toolkit.

Reserved Method Names

Some Java classes and/or objects have fields or methods whose names happen to be Obol reserved words. For example, the `Matcher` class has method `end()`. To access such a method simply quote its name as an Obol string. For example

```
let m = pattern.matcher(string) in m."end"();
```

Fly in the Ointment!

Notice that a `Button` object is composed of a Java core object extended with an Obol object. Such Obol extensions of Java objects are called *Hybrid* objects, and have the important limitation that calls from the Obol object to the core Java object must be super-calls.

Hybrid objects present two faces: one to the Java world and one to the Obol world.

When either their Java or their Obol methods are invoked from Obol, the right thing happens – overriding of Java methods by Obol methods works as it would in a non-hybrid object.

But the methods of a hybrid that are normally invoked from Java[24] are always the methods of the Java core object. In short, *one can extend a Java object in Obol for use by Obol, but one cannot do so for use by Java.*

This might seem to strike the death-knell for graphical objects, since they are nearly always implemented by overriding the `paint` method of something like a `Canvas`. Well, nearly! Happily it is a very simple matter to implement a suitable Java adapter class.

[24] *i.e.* by Java code written without special knowledge of the Obol machinery.

Here, for example, is the essence of the definition of `obol.Canvas` (from the inbuilt Java library).

```
package obol.awt;
import java.awt.*;

public class Canvas extends java.awt.Canvas
{
  Painter p;
  public Canvas setPainter(Painter p)  { this.p=p; return this; }
  public void paint(Graphics g)
  { if (p!=null)
       p.paint(g);
    else
       raise new Error("obol.canvas without Painter");
  }
}
```

An `obol.Canvas` paints with an `obol.Painter`

```
package obol.awt;
import java.awt.*;
public interface Painter { public void paint(Graphics g); }
```

Next we construct a very small Obol application that illustrates one way of constructing and using a canvas. First we define a `Canvas` procedure whose argument should be a procedure of type `java.awt.Graphics->{}` that does the painting.

```
let owt = package("obol.awt")
in  let Painter paintOn =
        owt.Painter.new {| paint gr = paintOn gr |}
    and Canvas paintOn =
        owt.Canvas.new().setPainter(Painter paintOn);
```

The application builds a fixed-size canvas, on which it writes ... the usual shibboleth!

```
let app   = java.awt.Frame.new("Hello World")
and hello = Canvas paintOn where paintOn g =
            { g.setColor(java.awt.Color.RED);
              g.setFont(java.awt.Font.decode("sanserif-bold-20"));
              g.drawString("Hello, World", 20, 20)
            }

in { app.add(hello);
     hello.setSize(200, 30);
     app.pack();
     app.show()
   }
```

3.2 Extracts from a Windowing Toolkit

Below we show an extract from the Obol windowing toolkit.

```
let owt =
module
  let obol       = package "obol"
  and Painter    = obol.awt.Painter.new
  and Canvas arg =
      obol.awt.Canvas.new arg with
      {|
         withFunctions obj =
         /*
             withFunctions
             {|  mousePressed  button (x, y) = ...;
                 mouseReleased button (x, y) = ...;
                 mouseMoved    button (x, y) = ...;
                 mouseDragged  button (x, y) = ...;
             |}

             Associates the given methods with the appropriate
             events on the underlying canvas. Other events
             on the canvas can still be listened for.
         */
         let funs = mouseFunctions obj in
         { super.addMouseListener(mouseListener(funs));
           super.addMouseMotionListener(mouseMotionListener(funs));
           this
         };
         withPainter obj = { super.setPainter(obj); this };
      |}

  and mouseFunctions obj =
  /*
      Yields an obol object, intended to be the argument for
      mouseListener or mouseMotionListener, (or both) that specifies
      responses to Pressed, Released, Moved, and Dragged events,
      and ignores all others.
  */
      let mouseResponse f e   =
          let p = e.getPoint() and b = e.getButton()
          in  f b (p.x, p.y)
  in
  {| mousePressed  e = mouseResponse (obj.mousePressed)  e;
     mouseReleased e = mouseResponse (obj.mouseReleased) e;
     mouseMoved    e = mouseResponse (obj.mouseMoved)    e;
     mouseDragged  e = mouseResponse (obj.mouseDragged)  e;
     invoke_       e = {}
  |}

  and reportObj = {| invoke_ it = stderr.println it |}

  and actionListener      = java.awt.event.ActionListener.new
  and actionFunction f    = actionListener
      {| actionPerformed e = f(e.getActionCommand()) |}
  and mouseListener       = java.awt.event.MouseListener.new
  and mouseMotionListener = java.awt.event.MouseMotionListener.new
  and Frame(title) = java.awt.Frame.new(title)
  and Label(text)  = java.awt.Label.new(text)
  and Button(text) = java.awt.Button.new(text) with
      {| withFunction(f) = { super.addActionListener(actionFunction(f)); this } |}
end;
```

Here is a small application that uses the library on the previous page. It makes a rectangle jump to the cursor when the mouse button is pressed, and follow the cursor when the mouse is dragged.

```
import owt;
let MouseCanvas(w, h) = import math in
    let x      = 0
    and y      = 0
    and canvas = Canvas()
    and functions =
        {|
           mouseDragged  b (_x, _y) = { x, y:=_x, _y; canvas.repaint(); };
           mousePressed  b (_x, _y) = { x, y:=_x, _y; canvas.repaint(); };
           mouseReleased b (_x, _y) = { };
           mouseMoved    b (_x, _y) = { };
        |}
    and painter = Painter
        {| paint g =
           {
             g.setColor(java.awt.Color.RED);
             g.fillRect(x, y, 50, 50);
             g.setColor(java.awt.Color.BLACK);
             g.drawLine(0, 0, x, y);
             g.drawLine(x, y, x+50, y+50);
             g.drawLine(canvas.getWidth(), canvas.getHeight(), x+50, y+50);
           }
        |}
    in
    { canvas.withPainter   painter;
      canvas.withFunctions functions;
      canvas.setSize(w, h);
      canvas.createBufferStrategy(1);
      canvas
    };

let app = Frame("Mouse Tracker")
in  { app.add(MouseCanvas(600, 400));
      app.pack();
      app.show();
      app.setLocation(200, 200)
    }
```

✂ ✂

A Note on Performance

Performance

Computational performance of the Obol interpreter as a whole is between 60% and 80% of the speed of compiled Python, without our having tried to optimize the interpreter in any way. It is also set to improve dramatically when we add "just-in-time" compilation. But it has to be said that Obol-to-Java (and Java-to-Obol Proxy) calls are not particularly fast[25] This is because of the intrinsic limitations of the Java reflection machinery that has to be used to implement such cross-language calls. Despite this speed limitation, cross-language calls are perfectly satisfactory for responding to `java.awt` events at user-interface speeds.

We expect that the main use to which the Obol to raw-Java interface will be put will be as a "glue" language – joining visual interfaces (also composed in Obol) to the applications they control. The applications themselves can be built in pure Java, or a mixture of Java and Obol: code with high performance requirements can migrate into Java as it is identified.

[25] Unless the Java side code uses the Obol extension machinery (which has yet to be documented).

A Built-in Modules

The following tables describe the constants, functions, and nested modules defined by Obol's built-in modules. The `global` module is special because its named values are always in scope (except in regions of program text where a nested declaration "masks" them). Moreover any declarations made at the top level of the interactive system are incorporated in it.

Module	Name and Type	Description
Error: Could not find or load main class oboldoc.ObolDoc		

B Obol Types

For any operator $\oplus$, the expression $e_1 \oplus e_2$ is evaluated by evaluating e_1 and e_2 then invoking the $\oplus$ method of the value of e_1. In the following table the value of e_1 is denoted *this* and that of e_2 is denoted *arg*.

Similarly $\oplus e$ is evaluated by evaluating e then invoking the (unary) $\oplus$ method of its value.

B.1 Built-In Types

this Type	Method Type.	Description
Error: Could not find or load main class oboldoc.ObolDoc		

B.2 Extension Types used in the Java interface

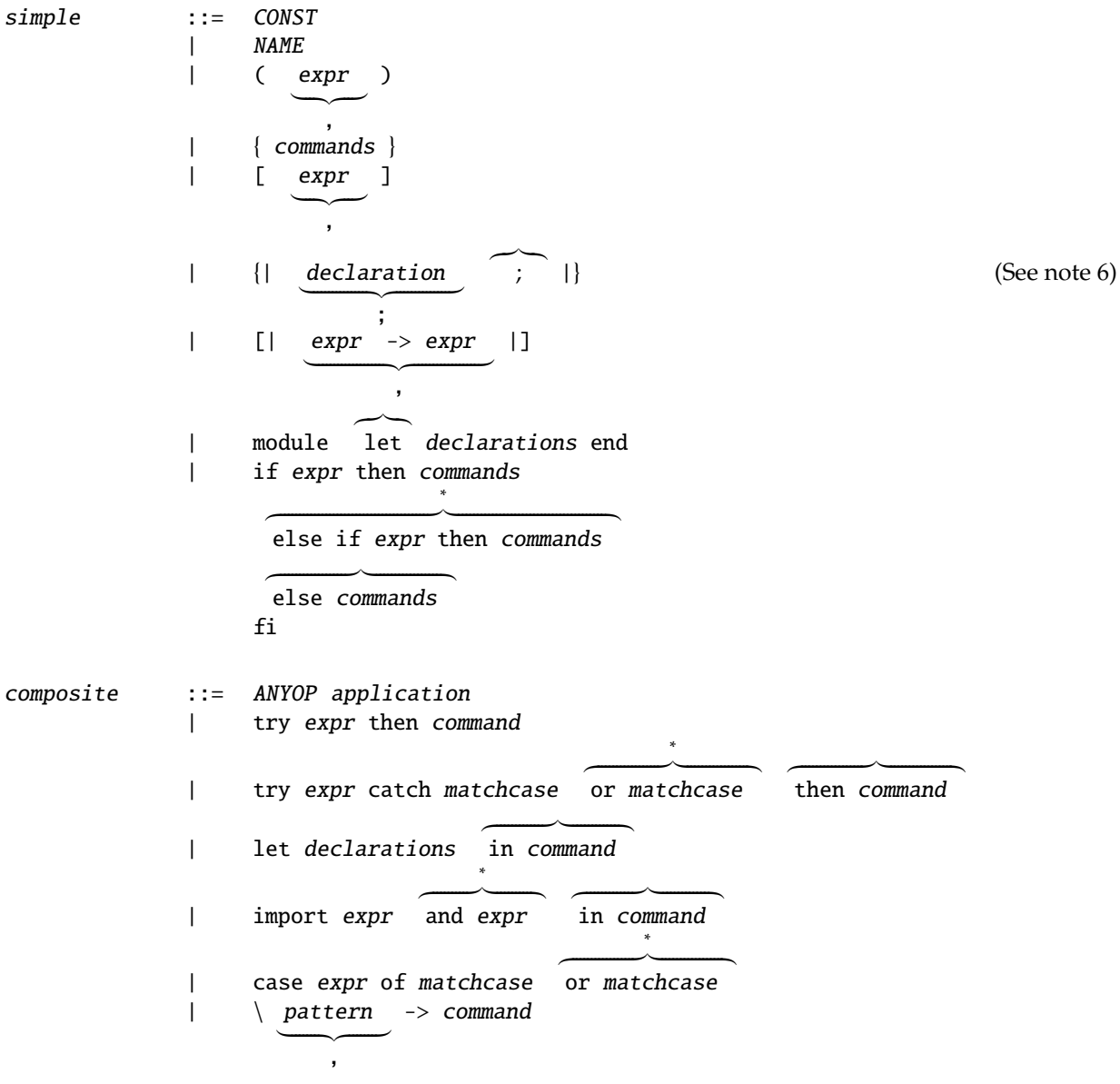
✂ ✂

Error: Could not find or load main class oboldoc.ObolDoc

C Obol Concrete Syntax

Here we specify, concisely, the language that is acceptable to the Obol parser. Not all sentences of this language are semantically meaningful. Some restrictions are explained in the body of the manual; others are explained in the notes that follow.

```
phrase       ::=  command
command      ::=  while expr do command
              |   for pattern in expr do command
              |   for pattern in expr while expr do command
              |   expr (,expr)* := expr (,expr)*                    (See note 1)
              |   expr                                              (See note 6)

expr         ::=  conditional (:typeexpr)  (where declarations)     (See note 2)

conditional  ::=  disjunction (=> conditional , conditional)*

disjunction  ::=  conjunction (|| conjunction)*

conjunction  ::=  relation (&& relation)*

relation     ::=  sum (RELOP sum)*                                  (See note 3)

sum          ::=  prod (ADDOP prod | with prod)*

prod         ::=  factor (MULOP factor)*

factor       ::=  application (EXPOP application)*

application  ::=  simple (. NAME | simple)*                         (See note 6)
              |   composite

declarations ::=  declaration (and declaration)*

commands     ::=  command (; command)* (;)*                         (See note 6)
declaration  ::=  (pattern ,)* pattern = command                    (See note 4)

matchcase    ::=  pattern -> command
pattern      ::=  expr                                              (See note 5)
simplepat    ::=  simple                                            (See note 5)
typeexpr     ::=  expr                                              (See note 2)
```



(See note 6)

Notes

1. The notation $\overbrace{\cdots}^{*}$ means zero or more occurences of $\cdots$. The notation $\overbrace{\cdots}$ means zero or one occurences of $\cdots$. The notation $\underbrace{\cdots}_{punct}$ means a sequence of one or more occurences of $\cdots$, separated by the *punct* symbol. The parser takes the "longest possible sequence" interpretation of all optional and repeatable forms.
2. The symbol -> is treated as an ADDOP when parsing type expressions. Type annotations are presently ignored by Obol's *semantic* processors.
3. The relation expression $e_1\ R_1\ e_2\ R_2\ e_3\ R_3 \cdots$ is short for $(e_1\ R_1\ e_2)\&\&(e_2\ R_2\ e_3)\&\&(e_3 \cdots)$

4. The function/method declaration *NAME simplepat simplepat ... = command* is syntactic sugar for *NAME* = *simplepat* -> *simplepat* -> ... *command*

5. An expression is only *semantically* acceptable as a pattern if it has one of the forms listed in the left hand column of the table in Appendix E.

6. Obol has a *forgiving* attitude to semicolons. This means that that the semicolons can (usually)[26] be omitted between the commands of a command sequence and between the declarations of a declaration sequence in a record object.

   The price to be paid for this is that all expressions are taken to end at the end of a line unless the last symbol on that line is a prefix or infix operator, or the first symbol on the following line is an opening bracket of some kind, *i.e.* one of the symbols `( { {| [ [|`.

   For example, the following program is written in Obol's forgiving syntax.

```
let gcd(m, n) =
    { while m!=n do if m<n then n:=n-m else if n<m then m:=m-n fi
      m
    }

let rat(n, d) =
    let f = gcd(n, d) in
    {| num = n/f
       den = d/f

       mul(other) = rat
                    (this.num*other.num, this.den*other.den)

      _print() = "%/%"%(num, den)
    |}

let a = rat(1,2)
let b = rat(4,5)
let _ = println(a.mul b)
```

[26] The only time when a semicolon is *required* between a command/declaration and the following command/declaration is when the second of them starts with an opening bracket of some kind.

CONST	::=	*NUMBER* | *STRING* | *CHAR* | ‘*NAME*	
RELOP	::=	< | <= | == | === | != | !==	
	|	> | >= | ~= | ~== | ! | ~	
ADDOP	::=	+ | - | & | vertical bar | -> | ~>	
	|	>> | << | ^ | ++	
MULOP	::=	* | / | % | <> | ><	
ANYOP	::=	*ADDOP* | *MULOP* | *RELOP* | *EXPOP*	
EXPOP	::=	# | ## | **	
NAME	::=	A sequence of letters, digits, or underscores, starting with a letter or underscore	(See note 1)
	|	?	
NUMBER	::=	A real number: a sequence of one or more digits, followed by a period then zero or more further digits, possibly followed by the letter `E` and one or more further digits.	
	|	A decimal integer: a sequence of decimal digits that does not begin with 0.	
	|	An octal integer: a sequence of decimal digits that begins with `0`.	
	|	A hexadecimal integer: a sequence of hexadecimal digits prefixed with `0x`.	
STRING	::=	" $\overbrace{\textit{CHARDES}}^{*}$ "	
CHAR	::=	’*CHARDES* ’	
CHARDES	::=	any single character except \, " or ’	
	|	\’	apostrophe
	|	\"	doublequote
	|	\\	backslash
	|	\r	return
	|	\n	newline
	|	\t	tab
	|	\f	formfeed
	|	\b	backspace
	|	\three octal digits	

Notes

1. The names `?, this, super, outer, global` are treated in most respects like ordinary variables, but may not be re-bound in patterns, or as formal parameters of functions or methods. The name _ is the “don’t care” pattern within programs. At the top-level of the interpreter it gets bound to the value of the last phrase to be successfully evaluated.

D Gotcha!

In this section we outline some frequently-occuring circumstances that may give rise to mystifying reports.

D.1 Syntactic

1. **Omitting Semicolons** Adjacent commands in a sequence of commands (or declarations in a sequence of declarations) can be separated by newlines instead of by semicolons. A command can be split across lines if the last symbol on the first line is an operator (infix or prefix) or the first symbol on the second line is an opening bracket or parenthesis of some kind. In the latter case Obol assumes that the first expression on the second line is the parameter of an application of a method/function/procedure, and warns about a possible missing semicolon.

   Example:

   ```
   >> let f x = x;
   >> { f
        (3)
      }
   Warning: possible omitted semicolon: (location in the source text)
   ```

2. **Where did my `where` go?** The expression *expr* **where** *declarations* is syntactic sugar for **let** *declarations* **in** *expr*. When the dynamic context of an error is being printed, the "unsugared" expression is shown – and you see a let clause rather than what was originally written.

   ```
   >> f(x where x=4);
   Error: variable undeclared: f
   at f(let x=4 in x)
   ```

3. **Application is more binding than Selection** The expression $f\ x.y$ is parsed as $(f(x)).y$. If you intend to apply f to $x.y$ make sure you parenthesise the operand, writing: $f(x.y)$.

4. **What happened to my declaration?** Do not use body-less declarations or imports within a sequence expression. They have no effect.

   ```
   >> {let x=3; ...; x}
   Error: x variable undeclared: x
   ```

   Solution: shift the declaration(s) to outside the sequence.

   ```
   >> let x=3 in { ...; x}
   ```

D.2 Semantic

1. **Attempting to redefine a built-in variable**

   If you attempt to redefine a built-in variable, the system will stop you.

   ```
   >> let cat = "cat"; println cat;
   Error: Cannot re-bind a global variable: cat to "cat"
   ```

   The solution is to restrict the scope of the redefinition to a module, and either qualify the name of the variable with the name of the module, or import the module.

   ```
   >> let m = module  let cat = "cat" end;
   >> println (m.cat);
   cat
   >> import m in println cat;
   cat
   ```

   Note that importing such a module "at the top level" with a body-less `import` also counts as a redefinition.

   ```
   >> import m;
   Error: Cannot re-bind a global variable: cat to "cat"
   ```

   This should not be too troublesome, since body-less imports can always be dispensed with.

2. **Accidentally iterating over a downward interval**

   This function is intended to be an array bubblesort

   ```
   let sort xs =
   { for i in 0 # (#xs-1) do
         for j in i+1 # (#xs-i-1) do
             if xs i > xs j then xs i, xs j := xs j, xs i fi;
     xs
   }
   ```

   If `#xs==0`, then `0 # (#xs - 1)` means `[0]` and the program will cause a subscript error.

   There are various solutions, the simplest of which is case analysis on *xs*.

   ```
   let sort xs = case #xs of 0->[] or _-> ... (as before) ...
   ```

E Pattern Matching

In the following rules $v, v_1, ..., w, w_1, ...$ denote variables; $p, p_1, ...$ denote more general patterns; $k, k_1, ...$ denote values in general; $c, c_1, ...$ denote simple values (numbers, strings, characters, constructors). In the bindings column, $p : k$ means "the bindings resulting from the match of p to k."

Pattern	Value	Succeeds if and only if	Bindings
v	k	always	v to k
_	k	always	None
c	c	always	None
()	()	always	None
[]	[]	always	None
{}	{}	always	None
`'tag` p	`'tag` k	p matches k	$p : k$
$(p_1, \cdots p_n)$	$(k_1, \cdots k_m)$	$m = n$ and all p_i match k_i	all of $p_i : k_i$
$[p_1, \cdots p_n]$	$[k_1, \cdots k_m]$	$m = n$ and all p_i match k_i	all of $p_i : k_i$
$\{\|v_1 = p_1; \cdots v_n = p_n\|\}$	$\{\|w_1 = k_1; \cdots w_m = k_m\|\}$	$\{v_1, \cdots\} \subseteq \{w_1, \cdots\}$ and every p_i matches the correspondingly-labelled k_j	all of $p_i : k_j$

The same variable may occur repeatedly in a structured pattern, but the pattern will only match the corresponding structured value if all occurences of the pattern match *equivalent* (not necessarily identical) values in the structure.

Examples

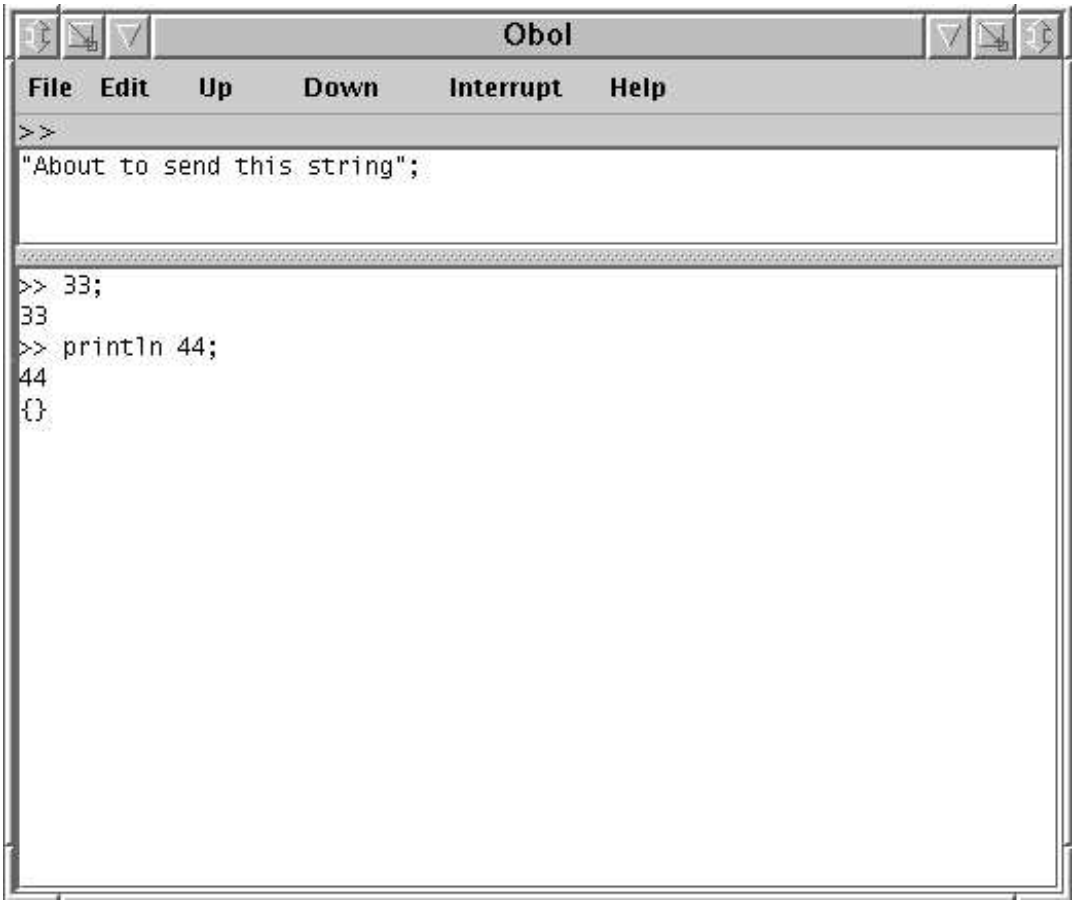
```
>> let s=1 in s;
1
>> let k = (1,2) in let (s,t)=k in (s, t);
(1, 2)
>> let k = (1,2) in let (s,s)=k in (s, s);
Error: s is already bound to a value that does not match 1
>> let k = (1,1) in let (s,s)=k in (s, s);
(1, 1)
>> let k = {| a=1; b=(2,3); c=4 |} in case k of {| a=s; b=t |} -> (s, t);
(1, (2,3))
>> let k = {| a=1; b=(2,3); c=4 |} in case k of {| a=s; c=t |} -> (s, t);
(1, 4)
>> let k = `record {| a=1; b=(2,3); c=4 |} in case k of `record {| a=s; c=t |} -> (s, t);
(1, 4)
>> let k = `record {| a=1; b=`tuple(2,3); c=4 |} in case k of `record {| a=s; b=`tuple t |} -> (s, t);
(1, (2,3))
>> let k = `record {| a=1; b=`tuple(2,3); c=4 |} in case k of `record {| a=s; b=t |} -> (s, t);
(1, `tuple(2,3))
```

The following example, uses "unordered pair" records to demonstrate the significance of our use of the word *equivalent* in the description of matching patterns with repeated occurences of variables.

```
>> let pair(aa,bb) = {| a=aa; b=bb; this===other = a==other.a&&b==other.b || a==other.b&&b==other.a |};
>> let k1 = pair(1,2) and k2=pair(2, 1);
>> k1===k2;
true;
>> let s, s=k1, k2 in s;
{| a=2; b=1; ...|}
```

F The Interactive Obol Window

If the Obol interpreter is started without any arguments, or if it is started with first argument `-w`, then it launches a two-panel interactive Obol window that looks something like this:



Obol phrases are edited in the editing panel (the topmost panel when the system starts), and when `Enter` is pressed *at the end of the text in the panel* that text is sent to the Obol interpreter. If the text formed a complete Obol phrase then it is evaluated and the resulting output is appended to the log panel.

The prompt ">>" is shown just above the top panel when the system is ready to read a new phrase. It is replaced by ">>>" when the system has read part, but not all, of a phrase, and by "`Evaluating`" while the system is executing a phrase.

The `Up` and `Down` buttons (and the corresponding keys) can be used to navigate in the history of the session.

Pressing the `Interrupt` button (or typing Control-C in the top panel) will interrupt the evaluation of the phrase that is currently being evaluated. This feature causes problems on some installations, so it has to be enabled by the `Interruptable` checkbox on the `File` menu.

The `Restart` button on the `File` menu starts a more-or-less completely new Obol interpreter.

The `Quit` button on the `File` menu closes down the window.

G Getting and Installing Obol

G.1 Availability

The Obol installer can be found at

```
ftp::ftp.comlab.ox.ac.uk/pub/Packages/JAPE/OBOL/ObolInstall.jar
```

It can be run on a Windows or a Unix machine, providing the java runtime environment has been installed first.

G.2 Windows installation

1. Ensure that java 1.4 (or later) has been installed on your machine.
2. Acquire (see above) `ObolInstall.jar`
3. Decide on the folder that will be used as the installation folder for Obol. We recommend that you make a new folder for this purpose.
4. *OPTIONAL:* **Move `ObolInstall.jar` to the installation folder.**
5. Double click on `ObolInstall.jar` in the installation folder. This should bring up a new window that belongs to the installation program.

   Unless you moved `ObolInstall.jar` to the installation directory in the previous step, you should click on the "Choose Folder" button in that window, and choose the installation folder.

   Click on the `Install` button in that Window. If all goes well a new icon will appear in the installation folder. That's the icon you click on to start the Obol interpreter. You can move it (or send it) anywhere in your windows filestore.
6. (If all doesn't go well, then on most Windows systems with java installed it is possible to start Obol with a doubleclick on the file `Obol.jar`)

G.3 Unix installation

1. Ensure that java 1.4 (or later) has been installed on your machine, and that the `java` binary directory is somewhere on your path.
2. Acquire (see above) `ObolInstall.jar`
3. Decide on the directory that will be used as the installation directory for Obol.
4. *OPTIONAL:* **Move `ObolInstall.jar` to the installation directory, and change directory to the installation directory.**

5. Run the command `java -jar ObolInstall.jar`. This should bring up a new window that belongs to the installation program.

   Unless you moved `ObolInstall.jar` to the installation directory in the previous step, you should click on the "Choose Folder" button in that window, and choose the directory.

   Click on the `Install` button in that Window. If all goes well a new executable file, `obol`, will appear in the installation directory.

6. (If all doesn't go well, then on most Unix systems with java installed it is possible to start Obol with the command `java -jar Obol.jar`)

Note 1 (Page 6) Matching Declarations
In general a declaration takes the form *pattern* = *expression* – the declaration succeeds, binding the variables in the pattern, providing that the expression evaluates to a structure that the pattern matches; it fails, raising a `bind` exception, if the pattern fails to match. Pattern matching rules are given in Appendix E

Note 2 (Page 8) Obol and Shell Scripts
You can use the Obol system in "shell scripts", that can be run as complete programs. Here's a Unix/Linux "Hello, World" shell script:

```
#!/bin/env obol
println("Hello, World");
```

When executing an Obol program configured as a shell script (*i.e.* with #! on the first line), the Obol interpreter automatically supresses the printing of *null*, so this program will print the specified text, and nothing more.

The Unix command `obol` is simply a shell script containing the text

```
exec java -jar -server path_to_the_Obol_jar_file/Obol.jar $*
```

The system takes the following arguments, which are inspected in succession from the left

`-w`	Use the interactive window.[27]
`-s`	Force evaluation in scripting mode.
`+s`	Forbid evaluation in scripting mode.[28]
`-h` *number*	Set the log panel buffer size to *number* lines.
`-f` *filename*	Input the named Obol source file.[29]
filename	Input the named Obol source file; stop inspecting arguments.[30]
`--`	As above, but take input from the terminal.[31]

When there are no arguments, the system puts an interactive window up.

[27] If `-w` appears, it must be the first argument.
[28] Even if the Obol source files looks like a script.
[29] The environment variable `OBOLPATH` is used as a search path when opening Obol source files.
[30] This and all subsequent arguments are handed to the Obol program in the variable `system.args`.
[31] ... or from the editing panel, if the interactive Obol Window is being used.

Note 3 (Page 14) Obol Exceptions
Although the simplest exceptions are represented as strings, Obol permits the throwing of other types of exception. These are delivered to the catch clause as a construction labelled with the data constructor `‘thrown`.

```
>> try throw 45 catch s -> s;
‘thrown 45
>> try throw ([3,5,7], 45) catch s -> s;
‘thrown ([3,5,7], 45))
```

If uncaught they are reported in the following form:

```
>> throw ([3,5,7], 45);
Error: thrown
 at throw([ 3, 5, 7 ], 45)
```

Subscript errors are normally reported in this form

```
>> [1,2,3](4);
Error: subscript
at [ 1, 2, 3 ](4)
```

but they can be caught. They are delivered to the catch clause as a construction labelled with the data constructor `‘subscript`.

```
>> try [1,2,3](4) catch s -> s;
‘subscript ([1, 2, 3], 4)
```

It is possible to discriminate on the content or structure of an exception, as the following examples illustrate:

```
>> let f x =
   case x of
      1 -> throw "one"
   or 2 -> throw ["two"]
   or 3 -> [1](2);
>> try f 1 catch ‘thrown x -> x or other->other;
"one"
>> try f 2 catch ‘thrown x -> x or other->other;
["two"]
>> try f 3 catch ‘thrown x -> x
   or ‘subscript (s, i) -> 1968
   or other->other;
1968
```

Running out of cases in a case expression causes a `case` error to be thrown. Here's an example.

```
>> try f 0 catch x -> x
‘case ("case x of 1 -> throw("one") or
                  2 -> throw["two"] or
                  3 -> ([1])(2) no cases match  0", 0)
```

Note 4 (Page 22) Records and Keyword Parameters
Records and record-patterns can be used to give the effect of "keyword-parameters":

```
>> let subst {| pat = regex; all = a; repl = r |} =
       let p = java.util.regex.Pattern.compile regex
       in  if a then
              \target -> p.matcher(target).replaceAll r
           else
              \target -> p.matcher(target).replaceFirst r
           fi;

>> subst {| pat="foo"; all=true; repl="baz" |} "foobaz is foo for you";
"bazbaz is baz for you"
>> subst {| pat="foo"; all=false; repl="baz" |} "foobaz is foo for you";
"bazbaz is foo for you"
```

Perhaps less obviously, record-extension can be used to give the effect of keyword parameters with defaults:

```
>> let substitute params = subst ({| all=true; repl="" |} ++ params);
>> substitute {| pat="foo" |} "foobaz is foo for you";
"baz is  for you"
```

Note 5 (Page 22) Record Equivalence
In this note we explain why Obol supports record-specific definitions of record equivalence.

Suppose we set out to represent rational numbers with records. It would make sense to build a rational-constructor procedure:

```
>> let rat(n, d) = {| num=n; den=d |};
```

It may or may not make sense for the implementer to insist that all rationals be kept in normal form (numerator and denominator coprime), but it is the implementer's decision. Were she to decide (perhaps for reasons of efficiency) not to do so, then the structural equivalence between rational representations would not concide with the semantics of rational numbers. For, although $rat(2, 4)$ and $rat(4, 8)$ represent identical rationals, they are not structurally equivalent.

```
>> rat(2,4)===rat(4,8);
false
```

This isn't really satisfactory – we want the equivalence test to be a *semantic* equivalence test. Happily Obol permits the designer of a record-based data type to define an equivalence procedure for that type. Anticipating, to some extent, our introduction to Objects, we give a glimpse of how this can be done for a (non-normalized) representation of rationals.

```
>> let rat(n, d) =
       {| num  = n;
          den  = d;
          this === other = this.num*other.den == this.den*other.num
       |};

>>     rat(2,4)===rat(4,8);
true
>>     rat(2,4)===rat(5,8);
false
```

Records representing rationals are now associated with an equivalence procedure, ===, that can be used as an infix operator between them and any other record with `num` and `den` fields.

Note 6 (Page 22) Structural Equivalence
Obol uses a structural equivalence procedure that is powerful enough to decide the equivalence of two finite, but posssibly cyclic, data structures – returning true if and only if the structures are isomorphic. The isomorphism is determined under the assumption that identically-named methods in distinct records are isomorphic if they have identical bound variables and bodies.[32]

In the following example `ones1` and `ones2` are distinct but isomorphic cyclic structures – representing an infinite list of 1s.

```
>> let ones1 = {| hd=1; tl={} |}; ones1.tl:=ones1;
>> ones1;
{| hd=1; tl=... |}

>> let ones2 = {| hd=1; tl={| hd=1; tl={} |} |}; ones2.tl.tl:=ones2;
>> ones2;
{| hd=1; tl={| hd=1; tl=... |} |}

>> equivalent(ones1, ones2);
true
```

In the following example, we define the core of a list-handling package, in which list equivalence is (almost)[33] structural equivalence.

[32] This assumption is necessary because (in general) the isomorphism of two procedures is undecideable.
[33] We insist on `nil` being a unique empty record, not just any empty record.

```
>> let cons(h, t) =
   {| hd=h;
      tl=t;
   |}
   and nil =
   {|
       this === other = other==nil
   |};

>> cons(1, cons(2, nil))==cons(1, cons(2, nil));
false
>> cons(1, cons(2, nil))===cons(1, cons(2, nil));
true
>> nil===={||};
false
>> {||}===nil;
false
```

✂ ✂

It is easy to experiment with tangled list structures. Below we define a function that replaces all occurences of `{}` in a cons-built structure with the root of the structure and construct a few tangled structures with it.

```
>>  let tangle root = { weave true root; root }
    where weave top cell =
    if cell==root && !top then
       {}
    else
    if isRecord(cell, "hd", "tl") then
       if isNull(cell.hd) then cell.hd:=root else weave false (cell.hd) fi;
       if isNull(cell.tl) then cell.tl:=root else weave false (cell.tl) fi;
    fi;

>> let a = tangle(cons(1, cons(2, cons(3, cons(1, cons(2, cons(3, {}))))))
   and b = tangle(cons(1, cons(2, cons(3, {}))))
   and c = tangle(cons(cons({}, {}), cons({}, {})))
   and d = tangle(cons({}, {}));

>> a===b;
true
>> c===d;
true
```

Index

`super`, 29
`this`
    considered un-necessary, 27
{}
    printing of, 5

Array, *see* Sequence
Assignment, 5
    simultaneous, 5
    unpacking, 5

Canvas
    Obol Implementation, 37
Case Expressions, 10

Data Constructors, 22
    in patterns, 22
Declaration
    matching, i
    unpacking, 6
Declarations
    persistent, 4
    scope of, *see* Scope
Dictionary
    mutable, *see* mutable mapping

Equality, 16
Equivalence
    of records, *see* Record
    of values, *see* Equality, Identity
Exceptions, 13
    discriminating on structure, ii
Expressions
    {}, 5

Forgiving attitude to semicolons, 46
Formatting, 32
Function
    `#`, 31
    `newArrayOf`, 33
    `newArray`, 33
    `newDict`, 19
    `newTableOf`, 33
    `newTable`, 33
Functions
    pattern-matching, 11

Hiding variables, 11
Hybrid Objects, 36

Identity, 16
Inheritance, 27
Installation
    Unix, 53
    Windows, 53
Iteration and Choice, 8

Java
    Class, 34
    Object, 34
    Package, 34
Java methods with reserved-word names, 36

Lambda Expressions, 12

Mapping
    immutable, 18
    mutable, 19
    notation, 18
Methods
    calls, 26
Modules
    Defining, 23
    Importing, 23
    Loading Code, 23
    parameterised, 24

Objects, *see* Records
    inheritance, 27
    prefix and infix operations on, 29
    super, 29
Obol
    command-line arguments, i
    in shell scripts, i
Operators, 29

Parameters

default, iii
keyword, iii
Patterns, 46
Performance, 40
Proxy
Obol, *see* Java
Java, *see* Java

Range, *see* Sequence
Range, 15
Record
Equivalence, 20
Extension, 21
Identity, 20
notation, 20

Scope
nested, 4
rules, 4
Sequence
Equivalence, 16
Identity, 16
immutable, 14, 15
mutable, 15
notation, 14
type of element, 17
Sequencing, 7
Structural Equivalence, iv

Table, *see* Sequence
Tuple, *see* Sequence
Type
annotations, 9, 45