Guide to Links

1. Running Links

To start an interactive shell:

1. Open the terminal.
2. Navigate to <path to Links>/bin.
3. Enter

```
./links
```

This will welcome you to Links. The list of all possible commands can be found in the appendix. **All** of the commands have to be finished with a **semi-colon**! Otherwise, you will see a line of dots:

```
. . . . . .
```

It means Links is expecting more to be typed.

The most important command is `@load "filename";`. It will compile the given file and either report any errors or will successfully finish and show the return value.

You can also enter

```
./links filename
```

in the terminal (no quotes around filename). It will be equivalent to

```
./links
@load "filename";
```

2. Syntax

2.1 General

- Comments are indicated by # and continue for the rest of the line.

- Literals are the following:

Integers: 2, 7 (type `Int`)
Floating point numbers: 14.5, 0.1 (type `Float`)
Booleans: true, false (type `Bool`)
Strings: “Rainbow”, “\”No way!\”, he said.” (type `String`)
Characters: 'a', '89', '\012' (type `Char`)

- Arithmetic operators for Float type are followed by a dot:

```
+     (Int, Int) → (Int)        Example:  2   +   2;
+.    (Float, Float) → (Float)  Example: 2.6  +.  9.1;
```

All standard operators:

```
+    -    *    /    ^   mod     for integers.
+.   -.   *.   /.   ^.          for floats.
```

These operators can be used infix as in examples, or prefix, if enclosed in parentheses:

```
(*.) (6.0, 7.3)
```

2.2 Lists

List is constructed using [] and :: . [] means empty list. :: adds an element to the start of the list. All elements of the list have to be of the same type. Example:

```
1 :: 4 :: 9 :: 16 :: []
```

The more convenient way, however, is to write it like this:

`[1, 4, 9, 16]` (is equivalent to the expression above)

```
[ ]
```

```
[“apple”, “strawberry”]
```

```
x = true
[true, x, false]
```

Adding an element to the start of the list:

```
1 :: [4, 9, 16] → [1, 4, 9, 16]
```

Concatenating two lists (uses ++ operator):

```
[1, 2] ++ [3, 4, 5] → [1, 2, 3, 4, 5]
```

Comparing two lists (uses ==):

`[4, 6] == [4, 6]` → true
`[“hello”, “world”] == [“HELLO”, “WORLD”]` → false

There are utility functions provided to work on the lists, such as, `hd` (for head), `take`, `drop`, etc. The full list can be found in appendix.

To match patterns on lists, a `switch` statement is used:

```
switch(list) {
     case [] -> specify actions if the list is empty
     case x::xs -> specify actions if there is an element.
}
```

`x` is the first element of the given list, whereas `xs` is the rest of the list. Examples can be found in appendix.

For integer lists it is possible to give range instead of writing out all the elements:

`[a..b]` , where `a < b`. For example:

`[4..9]` → `[4, 5, 6, 7, 8, 9]`
`[9..4]` → `[]`, because `a > b`.
`[4..4]` → `[4]`

2.3 Tuples

A tuple is like a list, but can contain elements of different types. An *n*-tuple is constructed by enclosing *n* expressions in parenthesis and separating them with commas:

```
(5, “Jazz”)
(false, 4.9, 'c')
```

The elements of a tuple can be accessed using pattern matching

```
var (x, y) = (5, “Jazz”)
x → 5
y → “Jazz”
```

You can also use dot notation or appropriate functions. For example:

```
(5, “Jazz”).1 → 5
second((5, “Jazz”)) → “Jazz”
```

The functions can be found in appendix.

2.4 Records

A record is similar to tuple, but rather than having integer indices, it retrieves the values by field names. Example:

```
(name=”Elvina”, university=”St Andrews”, country=”Lithuania”)
```

Field names (*name*, *university*, *country*) are not surrounded by quotes, but the values are. Field names also have to be constants at the time of constructing them.

`(name=”Elvina”, university=”St Andrews”)` → Correct.

`(first + name = ”Elvina”, university=”St Andrews”)` → Wrong.

`(“first” + “name” = ”Elvina”, university=”St Andrews”)` → Wrong.

The elements of the record are accessed using dot notation (also called *projecting*).

```
(name=”Elvina”, university=”St Andrews”).name == “Elvina”
```

Obviously, a record can be a variable, for example:

```
var item = (name=”Elvina”, university=”St Andrews”);
```

In which case the projection would look like this:

```
item.name == “Elvina”
```

To add a field to the existing record (assuming declaration above):

```
(favouriteColour = “pink” | item)
```

It would now yield:

```
(favouriteColour=”pink”, name=”Elvina”, university=”St Andrews”)
```

To overwrite a value in the existing field, use `with`:

```
(item with name = “Austeja”)
```

It now would yield:

```
(name=”Austeja”, university=”St Andrews”)
```

2.5 Comparisons and boolean expressions

Comparisons are binary operations that yield a true/false value.

- ==

True iff the operands evaluate to the same value.

- <>

True iff the operands evaluate to different values.

- <

True iff the left operand's value is less than the right operand's

- >

True iff the right operand's value is less than the left operand's

- <=

True iff the left operand's value is less than or equal to the right operand's

- >=

True iff the right operand's value is less than or equal to the left operand's

Boolean expressions can be combined using the Boolean operators:

- &&

- ||
- not

For example:

`not(true)` → false

2.6 Conditionals

Such expression has a condition, a consequent and an else clause. All three have to be present.

Correct:
```
if (x == y)
     expr1
else
     expr2
```

Wrong:
```
if (x == y)
     expr1
```

If expr1 or expr2 is made of more than one statement, they have to be surrounded by { }. If-else always returns a value and the return values' types must match.

Correct:
```
if (x == y)
     4
else
     9
```

Wrong:
```
if (x == y)
     4
else
     “Nine”
```

2.7 Variables

To declare a variable, use:

```
var name = value;
```

For example:

```
var item = “coffee”;
```

All variables have to have a value bind to them at the point of constructing.

`var name;` → Wrong.

Variable assignments have block scope. The following example

```
var x = 1;
if (condition) {
  var x = 2
} else {
  var x = 3
};
print(intToString(x));
```

prints 1 because the assignments to x within the if clauses only bind within those clauses. If you want the value printed to depend on the condition, you should assign to x from the result of the whole if expression:

```
var x = if (condition) {
  2
} else {
  3
};
```

2.8 Functions

Functions can be either named or anonymous. Named function has to start with `fun`, followed by a function name, brackets and parameters. Function body is opened and closed with curly braces. Example:

```
fun sum(x, y, z)
{
    # ... body
}
```

Anonymous functions only omit the name:

```
fun (x, y, z) { x + y + z }
```

A function is called using its name and listing the required arguments in parentheses:

```
sum(1, 2, 7)
```

It also is applicable even if a function is bound to a variable:

```
var sum = fun (x, y, z) { x + y + z };
sum(1, 2, 7)
```

Any expression that evaluates to a function value can be called:

```
if (true)
      fun (x) { x + 1 }
else
      fun (x) { x + 2 }
```

The return value of the function is indicated by a statement that does not have a semi-colon in the end. For example, this is correct:

```
fun add(x, y) {
     x + y
}
```

While this would be a type error:

```
fun add(x, y) {
     x + y;
}
```

If you want to return nothing, for example, print action does not need any return value, add unit ( ) in the end. This example is now correct:

```
fun add(x, y) {
     x + y;
     ()
}
```

2.9 Loops (List comprehensions)

Loop is the list comprehension in Links.

```
    for (x <- source)
       body
```

Both the source and the body should be expressions that evaluate to lists.

The value of a comprehension is the concatenation of all the lists produced by evaluating the body, once for each element of *source*, and binding that element to the variable x. For example:

```
  var source_list = [1, 2, 3];
    for (x <- source_list)
        [ x*x ]
```

constructs a list of the squares of the values in source_list. Note that more than one value can be included in the body list:

```
  var source_list = [2, 3, 7, 8, 9, 55];
    for (n <- source_list)
        if (odd(n))
           [n, n+1]
        else
           [n]
```

This example returns `[2, 3, 4, 7, 8, 8, 9, 10, 55, 56]`.

Filtering comprehensions

A comprehension can be filtered using the where clause.

```
var source = [2, 3, 4, 5, 6, 7];
  for (x <- source)
  where (odd(x))
    [x+1]
```

returns `[4, 6, 8]`.

A where clause is equivalent to a condition nested within a comprehension:

```
for (x <- src)
  where (condition)
    expr
```

is equivalent to

```
for (x <- src)
    if (condition)
      expr
    else []
```

`where` is a clause on for comprehensions: it cannot be used outside of a `for`.

Sorting comprehensions

The `orderby` clause on for comprehensions is used to sort the source before evaluating the body.

For example, suppose we have a list declared previously with type

`[(release_year:Int, model_number:Int, model_name:String)]`

describing models of an automobiles. Then the following will return a list of pairs describing the models, ordered by their year of release:

```
for (m <- models)
orderby (m.release_year)
    [(m.model_number, m.model_name)]
```

Multi-generator comprehensions

A comprehension can draw elements from more than one list. For each element produced by the first *generator*, Links iterates over all the items produced by the remaining generators.

For example:

```
for (fruit <- ["apple", "orange", "banana"], i <- [1..4])
        [(i, fruit)];
```

yields:

```
[(1, "apple"),  (2, "apple"),  (3, "apple"),  (4, "apple"), (1,
"banana"), (2, "banana"), (3, "banana"), (4, "banana"), (1,
"orange"), (2, "orange"), (3, "orange"), (4, "orange")]
```

You can also impose an order on all the elements produced by the series of generators in a comprehension header, as in:

```
  for (fruit <- ["apple", "orange", "banana"], i <- [1..4])
          orderby (fruit)
          [(i, fruit)];
```

yields:

```
[(1, "apple"),  (2, "apple"),  (3, "apple"),  (4, "apple"), (1,
"banana"), (2, "banana"), (3, "banana"), (4, "banana"),(1,
"orange"), (2, "orange"), (3, "orange"), (4, "orange")]
```

Links will produce a list of tuple elements as dictated by the generators, then sort them, and finally evaluate the body expression for each element produced. Note that it is the source elements, *not* the body elements, which are sorted.

The effect of multi-generator comprehensions is much like that of nested comprehensions: the comprehension

```
   for (fruit <- ["apple", "orange", "banana"], i <- [1..4])
       [(i, fruit)];
```

behaves just like this one:

```
   for (fruit <- ["apple", "orange", "banana"])
       for (i <- [1..4])
         [(i, fruit)];
```

But multi-generator comprehensions are different from the nested counterparts when it comes to clauses such as `orderby`. This is because the `orderby` clause sorts the list of tuples produced by all the generators in the *most recent* comprehension header. When using nested single-generator comprehesions, you are sorting one series of elements which is then collected by another comprehension, for a result than may not obey the desired ordering. For example,

```
for (fruit <- ["apple", "orange", "banana"])
            for (i <- [1..4])
            orderby (i)
              [(i, fruit)];
```

```
[(1, "apple"),  (2, "apple"),  (3, "apple"),  (4, "apple"),(1,
"banana"), (2, "banana"), (3, "banana"), (4, "banana"),(1,
"orange"), (2, "orange"), (3, "orange"), (4, "orange")]
```

3. Typing

Links is a strongly-typed, statically-typed programming language. This means that every value has a type, and every operation admits just certain types as arguments. The compiler will check that a program uses every value in a way consistent with a type. This way, for example, you can concatenate two lists but you can't concatenate two integers--it wouldn't make sense, and if you try to do it, the compiler will report an error up front. If a program passes the compiler, you can be certain that it doesn't have any type errors (unless there are any bugs in the compiler!)

Basic types were already introduced in 2.1. In addition to that, tuple's or record's type is indicated with comma separated base types in parentheses. For example:

```
(42, “Unicorn”) : (Int, String)
(name=”Elvina”, university=”St Andrews”):(name:String, university:String)
```

* String is not exactly a base type, but rather an alias for [Char]. Strings are concatenated with ^^ symbol.

Type annotations and signatures

Links can infer type information for any program you give it. This means that you typically don't have to declare any types. But if you get a type error, it may not point to the part of the code where you really made the mistake. When dealing with constants, the type is always obvious. Type annotation is more useful when dealing with functions, whose type may be inferred in a way that's not obvious. Suppose for example that you have a function concatenateLists that you know should always return a list of integers. You can tell the compiler this, and it will check that it is the case:

```
sig  concatenateLists : ([Int], [Int]) ~> ([Int])
fun concatenateLists(x, y) {
     x ++ y
}
```

To write a function signature, you need to follow this pattern:

```
sig function_name : (parameter type, …) ~> (returned value type)
```

Write `sig` followed by a function name and a colon. In the parenthesis write the types of the arguments or leave ( ) if there are none. Straight arrow (~>) here indicates the return.
Another example:

```
sig print : (String) ~> ( )
```

This function takes a String and returns a unit. A unit can be looked at as nothing. There is no usable return value.

Type aliases

Sometimes it is convenient to define an alias for a type in order to avoid having to write out the whole type every time it is used. For instance, suppose you want a type for representing colours:

```
[|Red | Orange | Yellow | Green | Blue | Indigo |]
```

Then you can write:

```
typename Colour = [|Red | Orange | Yellow | Green | Blue | Indigo |];
```

and following this declaration you can use Colour in place of `[|Red | Orange | Yellow | Green | Blue | Indigo |]`. For instance, you can now define functions with the following signatures:

```
sig showColour : (Colour) ~> String
sig invertColour : (Colour) ~> Colour
sig combineColours : (Colour, Colour) ~> Colour
```

4. Session types

Session types are used to describe communication protocol between 2 ends in such a way that its rules are enforced by the compiler. Session types consist of a label (choice) and the sequence of types of values to be sent or received. The most important point about session types is that they are dual (if one end expects to send a value, the other end expects to receive it). Therefore, once you write a type for one end, the other end will be the same, only with some syntax changes.

One end is offering, the other end is making a choice. You might think of this as client and server, but keep in mind that, depending on the situation, the roles can be reversed in the middle of the session and then later restored again.

To declare a session type, write:

```
typename Session_type_name = ... ;
```

Session type name has to start with the capital. Declaration has to end with a semi-colon. To declare the ‘server’ (offering end):

```
typename ServerSide = [&| Greet:!String.?String.EndBang, Count:!Int.?Int.EndBang
|&];
```

[ ] , & and | must always be there. Greet and Count are labels, separated by a comma. It means that the client can choose either. This is very important: it does not mean a client has to choose Greet and then Count. Comma means ‘or’. Exclamation mark ! means sending a value, question mark ? means receiving a value. They are combined by a dot – it indicates continuation. To finish the session, write EndBang (for the server, client has EndQuery). So what happens here? A client can choose either Greet or Count label. If they choose Greet, they send a string, receive a string and the session is over. That is, you cannot choose Greet or Count again, because this example is not recursive. In case of Count, client sends an integer, receives an integer and session ends.

Because the session types are dual, wherever you need to have the client type from this server type, you can simply enter ~ServerSide. Or you can declare client type explicitly (a recommendation for beginning).

```
typename ClientSide = [+| Greet:?String.!String.EndQuery, Count:?Int.!
Int.EndQuery |+];
```

Did you notice? Wherever we had ! in server, it became ? in client and vice versa. EndBang becomes EndQuery. Client side also has + (making a selection) instead of & (offering a choice).

We now have the session types, but how do we use them? Currently there are 2 types of syntax in Links: GV and CP.

GV
For the server:

```
sig serverFunction : (ServerSide) ~> EndBang
fun serverFunction(s) {

     offer(s) {
          case Greet(s) ->
                         var s = send(“Hello”, s);
                         var (response, s) = receive(s);
                         s

          case Count(s) ->
                         var s = send(14, s);
                         var (response, s ) = receive(s);
                         s
     }
}
```

The server return value will always be EndBang. ServerSide is a channel that has the session type declared above. We start with offering on channel s and then describe the possible choices and their consequences. Take a note that the channel s has to be rebind every time. `Send` sends a pair – value and the updated channel. `Receive` gets a pair – value and an updated channel. The final s (no semi-colon after!) indicates returning EndBang, in other words, ending the session.

For the client:

```
sig clientFunction : (~ServerSide) ~> ()
     or
sig clientFunction : (ClientSide) ~> ()

fun clientFunction(c) {
     var (response, c) = receive(select Greet c);
     var c = send(“Thank you for hello.”, c);
     wait(c);
     ( )
}
```

The client does not need to return the end of the session, it does so by `wait(c)` and then you can return a desired value to the caller function (in this case, return value is only a unit, but, for example, in calculator, you would return a sum of numbers you received from the server). Client operates in the same way as far as send/receive are concerned. The difference is that it has to `select` from the available choices and thus indicate how is channel c updated.

How do we initiate communication in the first place?

```
clientFunction(fork(serverFunction))
```

It spawns a separate proccess for the right hand side (serverFunction) and returns the value of the left hand side (clientFunction). You can stick it in main and then simply call main().

The full example would look like this:

```
typename ServerSide = [&| Greet:!String.?String.EndBang, Count:!
Int.?Int.EndBang |&];

sig serverFunction : (ServerSide) ~> EndBang
fun serverFunction(s) {

    offer(s) {
        case Greet(s) ->
                    var s = send("Hello", s);
                    var (response, s) = receive(s);
                    s

        case Count(s) ->
                    var s = send(14, s);
                    var (response, s ) = receive(s);
                    s
    }
}

sig clientFunction  : (~ServerSide) ~> String
fun clientFunction(c) {

  var (response, c) = receive(select Greet c);
  var c = send("Thank you for hello.", c);
  wait(c);
  response

}

fun main() {
  clientFunction(fork(serverFunction))
}

main()
```

CP
For the server:

```
sig serverFunction : (ServerSide) ~> EndBang
fun serverFunction(s) {
```

```
    <| offer s {
       case Greet -> s["Hello"].s(response).s[]
       case Count -> s[14].s(response).s[] } |>
}
```

As you can see, the syntax is more compact, but perhaps, not as intuitive. Offer is surrounded with <| and |> . The channel s is automatically rebind. Values inside [ ] indicate send. Values inside ( ) indicate receive. End of the session is indicated by s[]. The rest of the logic is like in GV.

For the client:

```
sig clientFunction  : (~ServerSide, !String.EndBang) ~> EndBang
fun clientFunction(c, return) {
  <| Greet c.c(response).c["Thank you for
hello."].c().return[response].return[] |>
}
```

The client takes in a communications channel and return value channel `return`. Syntax is similar to server, except couple of things: `c()` roughly corresponds to `wait(c)` in GV. `return[response]` is the same as `response` being the last statement of the GV function. `Return[]` is returning EndBang (different than in GV, in which we could return any value after the channel is closed).

Initiation:

```
run (fun(return){<| nu s.({serverFunction(s)}|{clientFunction(s,
return)}) |>})
```

The code is `run` by passing an anonymous function that takes in `return` channel. Inside of this function roughly does the same as fork – a new process is spawned, except this time the return value is bind to the right hand side.

The full example would look like this:

```
typename ServerSide = [&| Greet:!String.?String.EndBang, Count:!
Int.?Int.EndBang |&];

sig serverFunction : (ServerSide) ~> EndBang
fun serverFunction(s) {
    <| offer s {
       case Greet -> s["Hello"].s(response).s[]
       case Count -> s[14].s(response).s[] } |>
}

sig clientFunction  : (~ServerSide, !String.EndBang) ~> EndBang
fun clientFunction(c, return) {
  <| Greet c.c(response).c["Thank you for
hello."].c().return[response].return[] |>
}
```

```
fun main() {
  run (fun(return){<| nu s.({serverFunction(s)}|{clientFunction(s,
return)}) |>})
}

main()
```

This example in both GV and CP returns “Hello” in the end.

Session type following session type

Not always we want to give the client any choice. We want to alter available choices depending on what has happened before. To enforce the order, a session type inside a session type is written. Sounds complicated, but it really is not. For example, let’s extend the previous session type so that if you choose Greet, you can then only choose Talk:

```
typename ServerSide = [&| Greet:!String.?String.[&|Talk:!String.EndBang|&],
                          Count:!Int.?Int.EndBang |&];
```

How do we interpret this? !String.?String meant a string has to be sent, then a string has to be received – a dot indicates continuation, so this exact order is a must. Therefore, we simply add a dot and put another session type. Now this session type is a must too. Having chosen Greet, you can only choose Talk, receive a string from the server and the session ends. Count is not available after Greet. You can nest session types in session types as many times as you want. A full example of this can be found in 5. Examples and Mini Tutorials section.

Recursive session types

It would be useful if we could select something and be able to do it again until some other condition/selection is made. So we introduce recursive session types. They are declared like this:

```
typename ServerSide = mu a.[&| Greet:!String.?String.[&|Talk:!String.a|&],
                               Count:!Int.?Int.EndBang |&];
```

`mu a.` Indicates recursion. `a` can be any name. This declaration is put before the appropriate session type. In this case, the client can choose Greet or Count. If they choose Greet, they can only chose Talk. Notice the `a` in Talk label. It means having sent a string, the server goes back to the state where you can again choose either Greet or Count. Choosing Count at any point will end the session as it has EndBang. Recursion can be put before any session type and used at any point in further nested session types. An example can be found in 5. Examples and Mini Tutorials section.

Reversing client and server

As mentioned before, client and server can swap the roles. For example:

```
typename ServerSide = [&| Greet:!String.?String.
```

```
                                    [+|Talk:?String.EndBang, NotTalk:EndBang|+],
                          Count:!Int.?Int.EndBang |&];
```

Say the server receives Greet. Instead of offering something, it now selects – either Talk or NotTalk. The client on the other side selected Greet and then offers a selection of Talk and NotTalk. This can be useful for certain situations where the server could accept or reject something and the client would behave appropriately. An example of this can be found in 5. Examples and Mini Tutorials section.

Notes on CP

Sometimes you will need to do extra calculations or will simply want to print something immediately after you received it. To do that, open a block and when you need to continue the communication, open another <|. For example:

```
<| Greet c.c["Hello, server"].c(response).{print(response); <|
            c().return[()].return[] |> } |>
}
```

You might also need to call functions in the middle of the session that will continue with that session (especially applicable with recursion). To do that, you have to keep passing return channel, until you can finally end the communication. For example:

```
<| Greet c.c["Hello, server"].c(response).{continueSession(c,
return)} |>
```

Eventually, you will be able to write `c().return[#something].return[]` to end the session.

5. Examples and Mini Tutorials

Although Links does not have to have a function called “main”, but there has to be a main function that starts everything.

5.1 Hello world

```
0 sig myFirstProgram : ( ) ~> ( )
1 fun myFirstProgram( ) {
2     print(“Hello World!”)
3 }
4 myFirstProgram( )
```

returns:

```
Hello World!
() : ()
```

What happened here? Let’s go through steps. The signature (sig) tells us that function myFirstProgram will take no arguments and will return a unit (that is, consider it returns nothing usable). Squiggly arrow indicates that this function cannot be compiled to database – seems fair enough, it only contains a print statement. As a precaution, you can try always using squiggly arrows and then change back to straight arrows where Links allows to. Having done the signature, we actually write a function we just described. print takes in a String argument and will print it out to the console. Now if you look at Appendix, print returns a unit ( ). And we omitted a semi-colon in line 2, which means this is a returning statement. So the flow will go like this: print will return ( ), and myFirstProgram will return what print returns, because a semi-colon is missing. We could have also written the same function like this:

```
0 sig myFirstProgram : ( ) ~> ( )
1 fun myFirstProgram( ) {
2     print(“Hello World!”);
3     ( )
4 }
5 myFirstProgram( )
```

Links prints out the return value and its type, which in this case is ( ) : ( ). Don’t forget to call a function outside any other functions (line 5 in the latest example)!

5.2 Unwords

One of the most commonly used functions is `unwords`. It takes a list of Strings and concatenates it into a single one recursively.

```
0 sig unwords : ([String]) ~> (String)
1 fun unwords(list)
2 {
3         switch(list) {
4                 case [] -> ""
5                 case x::xs ->
6                             if (length(xs) == 0) x
7                             else x ^^ " " ^^ unwords(xs)
8         }
9 }
```

Let's assume this call somewhere in our program:
`unwords([“Shine”, “on”, “you”, “diamond”]);`

As before, we declare a signature which will ensure this function is used with correct types. To iterate through the list, switch statement is used and necessary cases are declared. In case of list being empty, an empty string is returned. Sounds fairly logical – if there is nothing to concatenate. Pay attention that there is no semi-colon as this “” value is the return value. Putting a semi-colon would be an error. In case there is at least one element (x::xs notation), we check, whether it is the last one with length of the rest of the list. Sounds complicated? Not at all. Again: if we reach x::xs point, we are sure x is some existing element. And xs is the rest of the list. If xs is empty, that means

x was the last element in the list and there is nothing else to be done, therefore we return x, which will be a concatenation due to the else clause. In else, if x is not the last element, we add it, concatenate with a space and make a recursive call to take the rest of the elements. Note the omission of semi-colons. The call will return:

```
"Shine on you diamond" : String
```

In a similar manner, various things can be applied/printed/added/etc. using `switch` on the lists.
* `^^` is an operator to concatenate strings.

5.3 Filter a list with a certain condition

The following function was designed as part of a solution to Kwic (Keywords in Context) problem. It is supposed to take a list of lists, where each inner list contains words of each title; a list with inconsequential words, such as “the”, “a”. It returns a first list, modified in such a way that each inner list that starts with any of the inconsequential words is removed. It is probably easier to see an example, supposed we have this:

```
[ [“Duck”, “Soup.”],  [“Soup.”, “Duck”], [“The”, “Wonders.”],
[ “Wonders.”, “The”] ];
```

and this:
```
[“the”, “of”]
```

The function would return:

```
[ [“Duck”, “Soup.”],  [“Soup.”, “Duck”], [ “Wonders.”, “The”] ];
```

Let's have a look:

```
0 sig filterWithInc : ([[String]], [String]) ~> ([[String]])
1 fun filterWithInc(list, excluded)
2 {
3       var excludedIgnoreCase = map(toUpperString, excluded);
4       for(x <- list)
5               where (not(elem(toUpperString(x !! 0), excludedIgnoreCase)))
6                       [x]
7
8 }
```

On line 3, we make a copy of excluded and turn everything upper case in order to avoid case sensitivity problems. We set up a loop – x is an element of a list. `for` will loop through all of the list automatically, while to achieve that with `switch`, a recursive function has to be made. We set up a filter – if the first element (x !! 0) of a list is not an element of `excluded`, add it to the final list. We again map to upper case in order to match line 3. It will go like this (case sensitivity has been dealt with):

`x == [“Duck”, “Soup”].x !! 0 ==` “Duck”. Is there “Duck” in `[“the”, “of”]`? No.

Let's take this whole list into a final list.

`x == [“The”, “Wonders.”]. x !! 0 ==` “The”. Is there “The” in `[“the”, “of”]`? Yes. Throw away this list.

In the end, we have a filtered list. List can also be filtered on elements being odd, even or any other necessary condition.

5.4 Session examples

5.4.1 Session type following session type

```
typename ServerSide = [&| Greet:!String.?String.[&|Talk:!
String.EndBang|&], Count:!Int.?Int.EndBang |&];

sig serverFun : (ServerSide) ~> EndBang
fun serverFun(s) {
  offer(s) {
      case Greet(s) ->
                      var s = send("Hello", s);
                      var (response, s) = receive(s);
                      offer(s) {
                        case Talk(s) ->
                                        var s = send("World", s);
                                        s
                      }
      case Count(s) ->
                       var s = send(14, s);
                       var (response, s) = receive(s);
                       s
  }
}

sig clientFun : (~ServerSide) ~> String
fun clientFun(c) {
    var c = select Greet c;
    var (response, c) = receive(c);
    var c = send("Thanks", c);
    var c = select Talk c;
    var (reply, c) = receive(c);
    wait(c);
    response
}


fun main() {
  clientFun(fork(serverFun))
}
```

```
main()
```

Returns “Hello”.

5.4.2 Recursion

```
typename ServerSide = mu a.[&| Greet:!String.?String.[&|Talk:!
String.a|&], Count:!Int.?Int.EndBang |&];

sig serverFun : (ServerSide) ~> EndBang
fun serverFun(s) {
  offer(s) {
      case Greet(s) ->
                      var s = send("Hello", s);
                      var (response, s) = receive(s);
                      offer(s) {
                        case Talk(s) ->
                                        var s = send("World", s);
                                        serverFun(s)
                      }
      case Count(s) ->
                       var s = send(14, s);
                       var (response, s) = receive(s);
                       s
  }
}

sig clientFun : (~ServerSide) ~> String
fun clientFun(c) {
    # Choose greet
    var c = select Greet c;
    var (response, c) = receive(c);
    var c = send("Thanks", c);
    var c = select Talk c;
    var (x, c) = receive(c);

    # Can choose greet or count
    var c = select Greet c;
    var (x, c) = receive(c);
    var c = send("Thanks again", c);
    var c = select Talk c;
    var (x, c) = receive(c);

    # Choose count and end
    var c = select Count c;
    var (x, c) = receive(c);
    var c = send(41, c);

    wait(c);
    response
```

```
}


fun main() {
  clientFun(fork(serverFun))
}

main()
```

Returns “Hello”.

5.4.3 Swap server and client.

```
typename ServerSide = [&| Greet:!String.?String.[+|Talk:?
String.EndBang, NotTalk:EndBang|+], Count:!Int.?Int.EndBang |&];

sig serverFun : (ServerSide) ~> EndBang
fun serverFun(s) {
  offer(s) {
      case Greet(s) ->
                      var s = send("Hello", s);
                      var (response, s) = receive(s);

                      # Will receive a value, because Talk.
                      var (response, s) = receive(select Talk s);
                      s

      case Count(s) ->
                       var s = send(14, s);
                       var (response, s) = receive(s);
                       s
  }
}

sig clientFun : (~ServerSide) ~> String
fun clientFun(c) {
    # Choose greet
    var c = select Greet c;
    var (response, c) = receive(c);
    var c = send("Thanks", c);


   offer(c) {
        case Talk(c) ->
                        var c = send("Talking", c);
                        wait(c);
                        response

        case NotTalk(c) ->
```

```
                          wait(c);
                          response
   }
}


fun main() {
  clientFun(fork(serverFun))
}

main()
```

Returns “Hello”.

Appendix

Links shell commands

The interpreter supports a number of directives. Typing one of these at the interactive shell has an immediate effect. End each of the commands with a semi-colon.

`@directives;`
list available directives
`@settings;`
print available settings
`@set;`
change the value of a setting
`@builtins;`
list builtin functions and values
`@quit;`
exit the interpreter
`@typeenv;`
display the current type environment
`@env;`
display the current value environment
`@load "filename";`
load in a Links source file, replacing the current environment
`@withtype;`
search for functions that match the given type

Library functions

To see all, enter `@builtins;` or `@typeenv;` in interactive Links shell.

Explicit type conversions

```
StringToInt        : (String) ~> Int
intToFloat         : (Int) ~> Float
intToString        : (Int) ~> String
floatToString      : (Float) ~> String
stringToXml        : (String) ~> Xml
intToXml           : (Int) ~> Xml
floatToXml         : (Int) ~> Xml
implode            : ([Char]) ~> String
explode            : (String) ~> [Char]
```

These function convert between values of different types.

```
ord             : (Char) ~> Int
chr             : (Int) ~> Char
```

The ord and chr functions converts between a character and its ASCII value as an integer.

Negation

```
not                 : (Bool) ~> Bool
negate              : (Int) ~> Int
negatef             : (Float) ~> Float
```

Character classification

```
isAlpha             : (Char) ~> Bool
isAlnum             : (Char) ~> Bool
isLower             : (Char) ~> Bool
isUpper             : (Char) ~> Bool
isDigit             : (Char) ~> Bool
isXDigit            : (Char) ~> Bool
isBlank             : (Char) ~> Bool
```

(isXDigit returns true for hexadecimal digits)

Case conversion

```
toUpper             : (Char) ~> Char
toLower             : (Char) ~> Char
```

Floating point functions

```
floor               : (Float) ~> Float
ceiling             : (Float) ~> Float
cos                 : (Float) ~> Float
sin                 : (Float) ~> Float
tan                 : (Float) ~> Float
log                 : (Float) ~> Float
sqrt                : (Float) ~> Float
```

Miscellaneous

```
print             : (String) ~> ()
```

Print a string to the standard output.

```
error             : (String) ~> a
```

Raise a fatal error.

```
debug             : (String) ~> ()
```

Output a string to stderr. When running a client function this will output to a special window only if links is run with the -d flag. When running a server function stderr is typically forwarded to the web server's error log.

```
sleep               : (Int) ~> ()
```

Wait for the specified number of seconds.

```
exit                : (a) ~> b
```

Exits the program, using the argument to exit as the program's return value.

List utilities

```
length : ([_]) ~> Int
```

Returns the number of elements in the given list.

```
all : (a b~> Bool, [a]) b~> Bool
```

Returns true if all elements of the list satisfy the predicate, false otherwise. (All elements of the empty list satisfy any predicate).

```
and : ([Bool]) ~> Bool
```

Returns true if all elements of the list are true.

```
any : ((a) b~> Bool, [a]) b~> Bool
```

Returns true if any element of the list satisfies the predicate, false otherwise. (No element of the empty list satisfies any predicate).

```
or : ([Bool]) ~> Bool
```

Returns true if any element of the list is true.

```
odd : (Int) ~> Bool even : (Int) ~> Bool
```

Returns true if the argument is, respectively, odd or even.

```
selectElem : ([a], Int) ~> a
```

selectElem(list, i) returns the ith element of the list, the first element being 0 (and throws an error if the list has fewer than i+1 elements).

```
swap : ([a], a, a) ~> [a]
```

swap(xs, a, b) takes each instance of a in xs and replaces it with b, and vice-versa, performing the two substitutions simultaneously.

```
fold_left : ((a, b) c~> a, a, [b]) c~> a
fold_right : ((a, b) c~> b, b, [a]) c~> b
fold_left1 : ((a, a) b~> a, [a]) b~> a
fold_right1 : ((a, a) b~> a, [a]) b~> a
```

The fold functions support arbitrary manipulation on lists in an ``algebraic'' way. In general, the caller gives a ``combining'' operation, an initial value, and a list. The fold combines each element of the list successively with the initial value to produce eventually a final value, which is the result of the fold. TBD

The expression fold_left(f, init, list) applies f successively to: the value init and the first element of the list, the result of that application and the second element of the list, the result of that next application and the third element of the list, and so on.

For example,

```
C<fold_left(f, init, ['a','b','c','d'])>
```

is equal to

```
C<f(f(f(f(init, 'a'), 'b'), 'c'), 'd')>
```

The fold_right function is similar but begins by applying f to the *last* element and init:

```
C<fold_right(f, init, ['a','b','c','d'])>
```

is equal to

```
C<f('a',f('b',f('c',f('d', init))))>
```

You can think of a fold as traversing a list with an ``accumulator'' whose value begins with the init value and is updated to the value produced by f working on the accumulator and the next element of the list as we traverse the input list.

The fold_left1 (resp. fold_right1) function behaves like its counterpart but uses the first (resp. last) element of the list as the initial value. This requires that the list be nonempty and the type of the initial value, result, and elements all be the same.

```
unzip : ([(a, b)]) ~> ([a], [b])
```

Split apart a list of paired elements into two parallel lists.

```
zip : ([a], [b]) ~> ([(a, b)])
```

Given two parallel lists, combine them into a single list of pairs, pairing the *i*the element of the first list with the *i*th element of the second list.

```
filter : (a ~> Bool, [a]) b~> [a]
```

Returns just those elements from the list that satisfy the given predicate.

```
compose : ((a) ~> c, (d) ~> a) ~> (d) ~> c
```

Given a function f and a function g, returns a function that applies g to its argument and f to the result of that.

```
id : (a) ~> a
```

Returns its argument.

```
map : ((a) ~> c, [a]) ~> [c]
```

Transforms a list by applying the function to each element of the list.

More precisely, it returns a list whose *i*th element is the result of applying the given function to the *i*th element of the given list.

```
concatMap : (a ~> [c], [a]) ~> [c]
```

Applies the function to each element of the list, and concatenates the resulting lists, thus returning a “flat”' list. Works much like a for loop in Links.

first, second, third, fourth, fifth, sixth, sevent, eighth, ninth, tenth

These functions each take a tuple of any (sufficiently large) size and return the first, second, etc. element of that tuple. For example:

```
links> third(4, 5, 6, 7, 8, 9)
  6
```

An attempt to apply one of these functions to a tuple that doesn't have enough elements will result in a type error.

```
sum : ([Int]) ~> Int
```

Returns the numerical sum of a list of integers.

```
product : ([Int]) ~> Int
```

Returns the numerical produce of a list of integers.

```
reverse : ([a]) ~> [a]
```

reverse(list) returns a list of the elements in list, but in the reverse order.

```
concat : ([[a]]) ~> [a]
```

Given a list of lists, this concatenates all of those lists together. E.g.:

```
  links> concat([1], [2, 3, 4], [], [5, 6])
    [1, 2, 3, 4, 5, 6]
```

```
join : ([a], [[a]]) ~> [a]
```

join(glue, list) returns a list formed by interspersing the ``glue" in between each element of list and concatenating them. E.g.:

```
  links> join(["/"], [["milk"], ["butter"], ["eggs"], ["bread",
"sand"]]);
   ["milk", "/", "butter", "/", "eggs", "/", "bread", "sand"] :
[String]
```

(Note the use of an ordinary String as a list of Chars.)

```
takeWhile : ((a) ~> Bool, [a]) ~> [a]
```

Returns the initial segment of the given list which satisfies the given predicate.

```
dropWhile : ((a) ~> Bool, [a]) ~> [a]
```

Returns all of the given list *but* the initial segment satisfying the given predicate. takeWhile(p,l) ++ dropWhile(p,l) is equal to l for any predicate p.

```
ignore : (_) ~> ()
```

Throws away its argument and returns unit, the empty tuple ().

```
isJust : Maybe (_) ~> Bool
```

Returns true if its argument is Just *x* for some *x*, that is, if it is not Nothing.

```
search : ((a) b~> Bool, [a]) b~> Maybe (a)
find : ((a) b~> Bool, [a]) b~> a
fromJust : Maybe (a) ~> a
```

Given an argument Just *x*, returns *x*. Otherwise, raises an error.

```
memassoc : (a,[(a, _)]) ~> Bool
```

Given a key and an association list, returns true if the key is in the domain of the association list.

```
lookup : (a, [(a, b)]) ~> Maybe (b)
assoc : (String, [(String, b)]) ~> b
assocAll : (String, [(String, b)]) ~> [b]
sortBy : ((a) ~> c, [a]) ~> [a]
```