# Miranda – Pure Functional Languages

## Miranda

Characteristics of Miranda:

- Purely Functional
- Lazy Evaluation
- Statically typed with Type Inference.

Haskell is similar to Miranda. It is more complicated and more widely used.

Drawbacks:

- Efficiency
- Acceptance
- Unnatural for some things: I/O and user interaction.



## Running Miranda

On educational Linux machines:

`% mira`

or

`% mira myfile.m`

This is the Miranda read-eval-print loop.



## Using `mira`

Miranda compiles functions to an intermediate representation.

While running `mira`:

`/edit` – Opens `emacs` with source file. When saved and quit, code gets recompiled and reloaded.
`/quit` – Quits `mira`.
`/help` – Prints `mira` command list.
`/aux` – Prints a different `mira` command list.
`?` – Prints list of all identifiers in global scope.
`/man` – Starts Miranda Manual (try item 28).

`<expr>` – Evaluates `<expr>` and returns its value.
`<expr> ::` – Evaluates `<expr>` and returns its type.

## Miranda Expressions

**function calls:** `<f-expr> <arg-expr-1> ... <arg-expr-k>`
For Example:

```
sqrt 81
reverse "hello"
map sqrt [1,4,9
```

**infix operators:** `<expr-1> <op> <expr-2>`
For Example:

```
5 + 10
5 > 10
1 : [5,6,7]
[1,2,3] ++ [4,5,6]
[0,1,2,3] ! 2
```

**infix operators as expressions:** `(<op>) <expr-1> <expr-2>`
Turn on infix operator into a 2 argument function with `(<op>)`.
For Example:

```
(+) 5 10
(:) 1 [5,6,7]
```

Use parentheses for grouping.



## Miranda Types

**num:** Either real or integer. E.g. `5` and `3.14`.

**char:** E.g. `'c'` and `'\n'`.

**bool:** E.g. `True` and `False`.

**lists:** Can be list of any single type.
E.g. `[True,True,False]` and `[1,2,3.14]` and `[[1],[2,3],[]]`.

**string:** Just a list of `char`. E.g. `['h','i']` or `"hello"`.

**tuples:** groupings of values.
E.g. `(1,"hi",True)` is of type `(num,[char],bool)`

**functions:** Functions are first class and have types.



## Type Exercise

What are the types of (or is it a type error?):

1. `(1,'a',3.4)`
2. `[1,'a',3.4]`
3. `[(1,'a',3.4),(2,'b',6)]`
4. `[[1,2,3],[]]`
5. `[[1,2,3],['a','b']]`
6. `["hi",['a','b']]`
7. `([1,2,3],['a','b'])`



## Lists

Like Scheme, Lists are building block for Miranda.
Creating lists:

```
[1,2,3]
[1..10]       || list of 1 to 10
[1..]         || infinite data structure
```

List Functions and operators:

`(:)` – the cons operator: `1 : [2,3,4]`.
`(++)` – the append operator: `[1,2] ++ [3,4]`.
`(!)` – the index operator: `[0,1,2,3] ! 2`.
`take k list` – first `k` elements of `list`.
`drop k list` – `list` with first `k` elements dropped.
`member list obj` – true if `obj` is in `list`.

## Functions

Define functions like:

```
add x y = x + y

add3 x y z = x + y + z
```

These can be used as:

```
add 5 10

add3 5 10 15
```



## Functions with *Guards*

(*Whitespace in function definitions is significant!*)

Instead of if-statements, use functions with *guards*:

```
my_abs x = x, if x >= 0
         = -x, otherwise          || note '=' are aligned.
```

Define `is_positive x` and `factorial n`



## Functions with *Patterns*

```
is_zero 0 = True
is_zero x = False                 || 'x' will match anything.
```

Notes:

- First pattern that matches is used.
- Patterns can contain variables.

```
my_and True x = x
my_and False x = False
```

Define `my_xor`



## More Patterns

Patterns on lists:

```
car (x:xs) = x
cdr (x:xs) = xs

length [] = 0                     || match empty list
length (x:xs) = 1 + length xs     || match non-empty list
```

Patterns on tuples:

```
sum_3tuple (x,y,z) = x+y+z

check_length (len,str) = len = length str
```

Patterns on numbers:

```
factorial 0 = 1
factorial (n+1) = (n+1) * factorial n
```

## Patterns on Lists

These are *very* useful.

Miranda provides `hd` and `tl` (like `car` and `cdr`). Compare:

```
length [] = 0                    || match empty list
length (x:xs) = 1 + length xs    || match non-empty list

length2 list = 0, if list = []
               1 + length (tl list), otherwise
```

Can pattern match on any constant number of items in list:

```
is_length_one [x] = True
is_length_one xs = False         || this is a "match all" pattern

is_one_two_three [1,2,3] = True
is_one_two_three xs = False

cadr (y:x:xs) = x
```



## Local Variables & Scoping

Miranda is lexically scoped like Scheme:

```
twice_reversed list = rev ++ rev
                      where rev = reverse list

|| make forward-forward-reverse-reverse... list
ffrrffrr list = ffrr ++ ffrr
                where ffrr = ff ++ rev ++ rev
                             where
                                rev = reverse list
                                ff = list ++ list
```



## Currying

Miranda functions are automatically curried:

```
add x y = x + y
add3 x y z = x + y + z
```

Consider:

1. `add 5 10`
2. `(add 5) 10`
3. `((add3 5) 10) 15`

Here `(add 5)` is the function that takes an number an returns than number plus 5.

Consider `plus5 x = add 5 x` or equivalently `plus5 = add 5` or even `plus5 = add3 1 4`

Can you redefine `add` using currying?



## Function Types

```
add x y = x + y
plus5 = add 5
```

What is the type of `plus5`?

It takes a number and returns a number. Type is: `num->num`

What is type of `add`?

## Curried Function Types

```
add x y = x + y
plus5 = add 5
```

Recall type of `plus5` is `num->num`.

What is type of `add`?

When `add` is passed `5` it returns `plus5`.

Type of `add` is `num->(plus5 ::)` which is `num->(num->num)`

Define `->` to be right associative.

`add ::= num->num->num`



## Curried Types (cont)

In general for :

`f x1 x2 ... xN = y`

We have:

`f ::= (x1 ::)->(x2 ::)->...->(xN ::)->(y ::)`

(Note: `y` may be an function)



## Curried Type Examples

Given:

```
foo 5 True 'a' = [1,2,3]
bar (2,'a') = add
baz 1 = foo
```

What are:

1. `foo ::`
2. `bar ::`
3. `baz ::`
4. `foo 5 True ::`



## Polymorphic Functions

Recall definition of `car`:

```
car (x:xs) = x
```

What is type of `car`?

## Type of `car`

```
car (x:xs) = x
```

We can use `car` as `car [1,2,3]` or `car [True,False]`.

Is `car` type `[num]->num` or `[bool]->bool`?

Actually `car` is type `[*]->*`.

Notice that `car` is a *polymorphic* function: it takes parameters of different types. E.g. `[num]` or `[bool]`.



## Higher-Order Functions

Miranda has `map`, `map2`, `foldr`, `foldl`, and `const` builtin.

For Example:

```
const x y = x

map2 f [] [] = []
map2 f (x:xs) (y:ys) = f x y : map2 f xs ys
```

Evaluate the following:

1. `map2 (+) [1,2] [3,4]`
2. `foldl (*) 1 [1,2,3]`
3. `foldr (:)  [4,5,6] [1,2,3]`
4. `const 5 False`
5. `foldr (+) 0 (map (const 1) [1..100])`

We can define our own:

```
twice f x = f (f x)
rev2 f x y = f y x
```



## Higher-Order Function Types

Recall:

```
const x y = x

map2 f [] [] = []
map2 f (x:xs) (y:ys) = f x y : map2 f xs ys
```

What are the values of:

1. `map2 ::`
2. `const ::`
3. `const 5 ::`
4. `map2 (+) ::`
5. `map ::`
6. `map (const True) ::`
7. `map2 (<) ::`



## Foldr and Foldl

```
foldr op id = f
              where
                 f [] = id
                 f (x:xs) = op x (f xs)

foldl op id = f id
              where
                 f id [] = id
                 f id (x:xs) = f (op id x) xs

|| these are standard
sum = foldl (+) 0
product = foldl (*) 1
or = foldr (\/) False
and = foldr (&) True
```

## Lazy Evaluation

Consider definitions:

```
my_if True x y = x
my_if False x y = y

ones = 1 : ones
```

Now run:

```
my_if (x>y) 3 4

my_if (x>y) 3 (1/0)

ones ! 1000
```



## Prime Number Generators

```
factors n = filter is_factor [1..n]
            where
               is_factor k = n mod k = 0

dullprimes = filter isprime [2..]
             where
                isprime p = (factors p = [1,p])

interestingprimes
   = sieve [2..]
     where
        sieve (p:x) = p : sieve filtered
                      where
                         filtered = filter not_multiple x
                         not_multiple n = n mod p > 0
```



## Hamming Numbers

Hamming numbers are $2^i 3^j 5^k$ for some $i, j, k$.

Lets generate an infinite list of Hamming numbers in order.

```
my_merge (x:a) (y:b) = x : my_merge a (y:b) , if x<y
                     = y : my_merge a b , if x=y
                     = y : my_merge (x:a) b , otherwise

ham = 1: my_merge ham2 (my_merge ham3 ham5)

ham2 = map (* 2) ham
ham3 = map (* 3) ham
ham5 = map (* 5) ham
```



## Tail Recursion

Another look at `foldl`:

```
foldl op id = f id
              where
                 f id [] = id
                 f id (x:xs) = f (op id x) xs
```

This is *tail-recursive* but Miranda will not optimize.

If we rewrite then Miranda will optimize:

```
foldl op id = f id
              where
                 f id [] = id
                 f id (x:xs) = f (op id x) xs, if id = id
```

Why will Miranda not optimize the first definition of `foldl`?