Type Inference

Do not need to provide miranda with types, but it is a good idea.

```
map::(*->**)->[*]->[**]
map f [] = []
map f (x:xs) = f x : map f xs
```

Whether you do or not, Miranda will infer the function type:
From line 1: `f::*`, `[]::[**]`, and `[]::[***]`
From line 2: `x::**` and `(f x)::***`

Thus: `f::**->***` so `map::(**->***)->[**]->[***]`

But we can rewrite using less `*`s:

`map::(*->**)->[*]->[**]`

Type Inference (cont)

Use type inference to find type of: `twice f x = f (f x)`

Currying & Type Inference

Consider: `map::(*->**)->[*]->[**]` and `neg::num->num`

What is `map neg ::`

Currying & Type Inference (cont)

Consider: `map::(*->**)->[*]->[**]` and `(+)::num->num->num`

What is `map (+) ::`

Type Aliases

Can create alias types:

```
string == [char]

arith == num->num->num
```


Tagged Types

Also called *algebraic types*:

```
weather ::= Rainy | Sunny | Cloudy | Snowy
month ::= Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | ...

forecast :: month->weather

forecast Aug = Sunny
forecast Jul = Sunny
forecast xxx = Rainy
```

Tag names must begin with a capital letter (variables must begin with lowercase letter).

`Rainy`, `Sunny`, etc. are constructors for type `weather`.

Tagged Types (cont)

Tagged values can contain variables (Product Types):

```
shape ::= Circle num | Square num |
          Rectangle num num | Triangle num num

pi = 3.14159

area :: shape->num
area (Circle r) = pi * r * r
area (Square width) = width * width
area (Rectangle width height) = width * height
area (Triangle base height) = base * height / 2.0

simplify :: shape->shape
simplify (Rectangle w w) = Square w
simplify s = s
```

These constructors are like functions.
For example, `Triangle ::= num->num->shape`.

Polymorphic Tagged Types

We can also have polymorphic and recursive tagged types:

```
tree * ::= EmptyTree | Node * (tree *) (tree *)

t = Node 7 EmptyTree (Node 4 EmptyTree EmptyTree)

flatten EmptyTree = []
flatten (Node n left right) = flatten left ++ [n] ++ flatten right
```

What is `t ::`

What does `flatten t` return?

What is `Node ::`? How about `Node 5 ::`?

List Comprehensions

Provide handy notation for constructing lists from other lists.

Recall set notation: $\{x \mid x \in A \text{ and } x > 0\}$

Miranda has a very similar notation: `[ x | x <- list; x > 0 ]`

This is read: "`x` such that `x` comes from `list` and `x > 0`."

```
[x | x <- [1..10]; x mod 2 = 0]     || evens between 1 and 10
[x*2 | x <- [1..5]]                 || evens between 1 and 10

[x^3 | x <- [1..]]                  || all cubes.
```


List Comprehensions (cont)

Redefine `filter` and `map`

```
filter' predicate list = [ x | x <- list; predicate x ]

map' f list = [ f x | x <- list ]
```

Can use patterns! If patterns do not match, item is discarded.

```
shapes = [Circle 5, Square 10, Rectangle 1 2, Circle 2]

radiuses list = [r | (Circle r) <- list]
```


List Comprehensions (cont)

Lets write quicksort with list comprehensions:

More examples:

```
|| produce all pairs from [1..5] and "abcd"
[ (a,b) | a <- [1..5]; b <- "abcd"]

|| same but diagonalized, for infinite lists
[ (a,b) // a <- [1..5]; b <- "abcd"]
```


Abstract Data Types

```
abstype queue *
    with emptyqueue :: queue *
         isempty :: queue * -> bool
         head :: queue * -> *
         enqueue :: queue * -> * -> queue *
         dequeue :: queue * -> queue *

queue * == [*]
emptyqueue = []
isempty [] = True
isempty (a:x) = False
head (a:x) = a
enqueue q a = q ++ [a]
dequeue (a:x) = x
```

Using the `queue`

Can use `queue` as follows:

```
q1 = enqueue (enqueue emptyqueue 1) 2

q2 = dequeue q1

x = head q1
```

But not:

```
q3 = dequeue [1,2,3]
```