Extensions for Miniml

Lexical environment based model

I made a lexical based model for evaluators which implement a lexically scoped environment semantics.

The evaluator is called eval_l.

About lexical scope

Ocaml is a lexically scoped language which means that any values are evaluated in the scope of the environment they are defined in, that is, it is governed by the lexical structure of the .

The substitution model by virtue of its definition is lexically scoped but the dynamic semantic does not. With dynamic syntax, the values determined by the dynamic ordering in which they are evaluated.

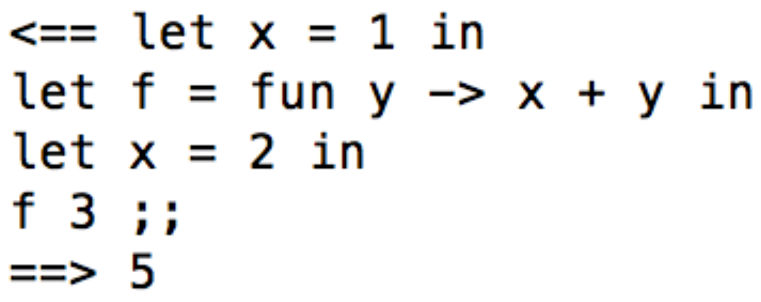
For example in the code below

```
let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
```

The function f is applied to three when the environment has value 2 assigned to x but since the function was defined in when x was equal to 3, a lexically scoped evaluator would scope f to x with value 1 resulting in a final value of 4. However, a dynamically scoped evaluator would calculate f with x of value 2 resulting in a final value of 5.

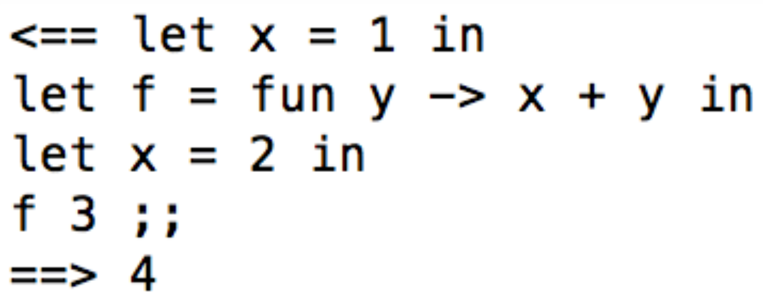
Below are some snapshots of the values outputed by eval_d and eval_l:

eval_l evaluates to 5



```
<== let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
==> 5
```

eval_d evaluates to 4



```
<== let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
==> 4
```

Implementation

Implementation

In order to implement lexical scoping, I changed the way functions (Fun) were evaluated so that whenever the evaluator comes across a Fun expression, it was made into a closure with the current (in the lexical scope of the function) environment. So when we need to evaluate an application, the function is now evaluated in the environment it was defined in and not the most recent dynamic environment.

After writing the eval_l evaluator, I realized that eval_l and eval_d have most of their code in common so I decided to make a new function called create_eval which takes an extra argument called model (which takes a model data type) in addition to those taken by eval_l and eval_d. Create_eval now creates a closure regardless of where the evaluator is lexically scoped. However, if the model parameter if provided as Dynamic, it makes a closure with an empty environment and does the application evaluation in the most recent environment.

I decided to use closure for both eval_d and eval_l since it made the match case in App evaluation simpler.

Additional Atomic Types

I implemented three additional atomic types: strings, floats and bignum.

In order to implement these I extensions, I looked up the information about doing parsing online and tried to understand the minimal_parse and minimal_lex files.

I then addes an extra regexp for the data types.

String

The regexp for string is

```
let strings = ['"'] [^ '"']* ['"']
```

Here, I match it with something that starts with a double quotation marks and ends with another double quotation. The values in between can be another other character other than double quotes.

The token for strings is STRING and it is represented as "String of string" in expr.

For Strings, I added extra match cases for the exp_to_string and exp_to_abstract_string functions.

The operations that can be done on Strings are Equal and Concat.

Concat is a new binop that I implemented which takes two "Strings" and concatenation.

In order to parse concat, I added an extra symbol '^' which is matched to a token CONCAT that lies between two other tokens.

The Concat expression is only applicable to strings and it raises expresssions for anything other than two strings.

Floats

The regexp for floats is

```
digit+ '.' digit*
```

This matches it with anything that has a period after more than or equal to 1 digit. The period may or may not be followed by other digits.

The token for floats is FLOATS and it is represented as "Float of float" in expr.

As for Strings, I added extra match cases for floats in the exp_to_string and exp_to_abstract_string functions.

The operations that can be done on floats are the same expressions that can be done on Num : Negate, Equals, LessThan, Plus, Minus, Times.

I decided to limit calculations for floats within floats themselves just like ocaml, so the unops and binops can only be implemented between floats and not between floats and integers.

Bignums

A common limitation we have with ints and floats is that they are limited by the memory size of their backend implementations so I decided to add an extension to my miniml which allows calculations for bignums just like the ones we implemented in PSet 3.

In implementing bignums I reused the code we wrote for PSet 3 with changes to reflect the comments I received for that pset.

In order to abstract the implementation of bignum, I implemented bignum as a module BN of type Bignum in the file bignum.ml.

The module Bignum has the signature as shown below.

```
module type Bignum = sig
    type bignum = {neg: bool; coeffs: int list} ;;
    val negate : bignum -> bignum
    val equal : bignum -> bignum -> bool
    val less : bignum -> bignum -> bool
    val fromString : string -> bignum
    val toString : bignum -> string
    val plus :  bignum -> bignum -> bignum
    val times : bignum -> bignum -> bignum
  end
```

The user can only access the functions made available by the signature while all other helper functions and backend implementations are abstracted away.

In order to parse bignums I made the regexp as

```
['B'] digit+
```

This regular expression matches with a token which starts with a capital B followed by one or more digits.

So 123456789 as a bignum is should be written as

```
B123456789
```

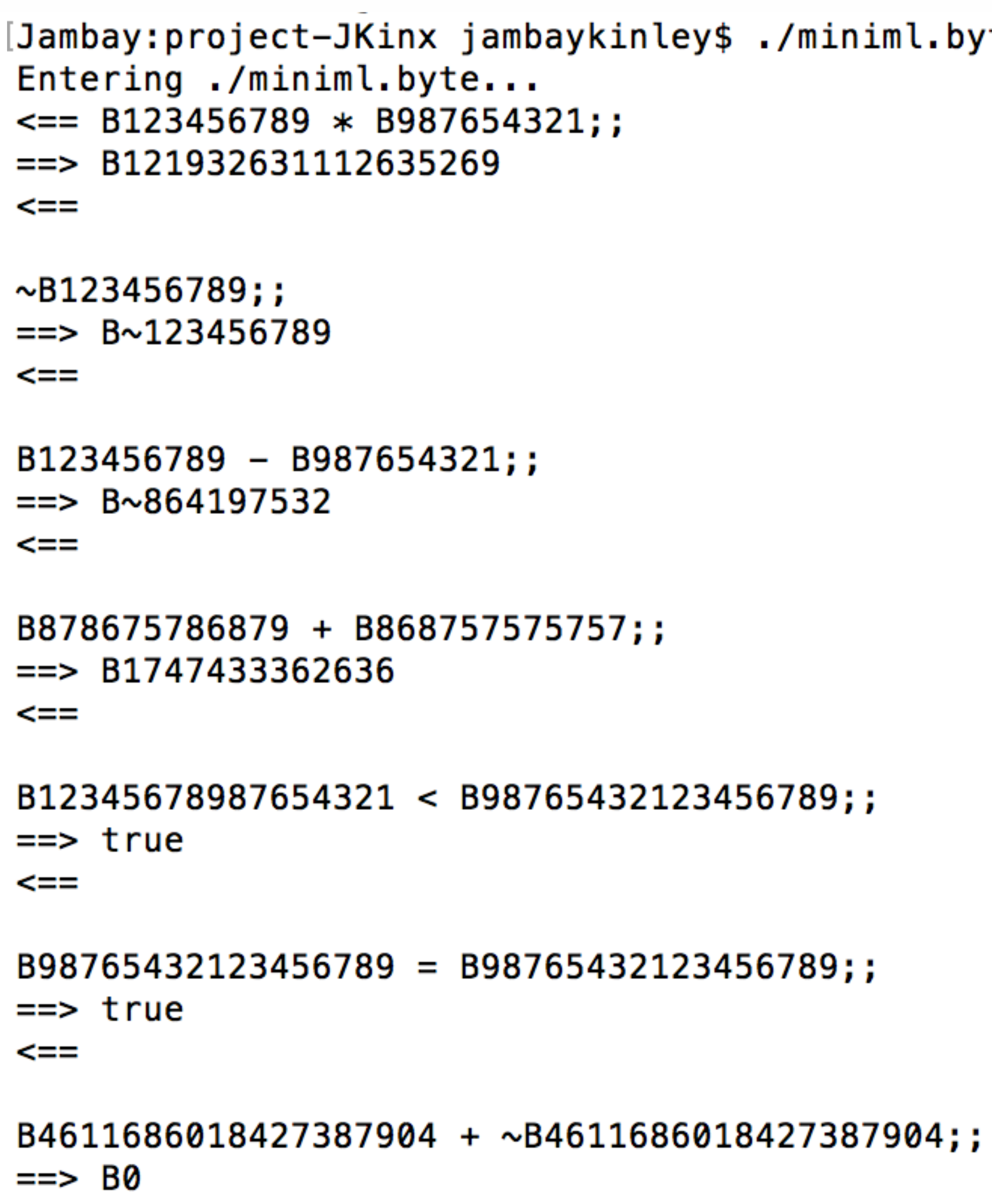
The token for bignums is BIGNUM and it is represented as Bignum of BN.bignum in expression.

As for Strings and Floats, I added extra match cases for floats in the exp_to_string and exp_to_abstract_string functions.

The operations that can be done on floats are the same expressions that can be done on Num and Ints: Negate, Equals, LessThan, Plus, Minus, Times.

Just like Floats and Nums, I decided to limit calculations for bignums within bignums themselves just like ocaml, so the unops and binops can only be implemented between bignums and not between bignums and anything else.

Some examples of using bignums in the miniml are shown below:



```
Jambay:project-JKinx jambaykinley$ ./miniml.byte
Entering ./miniml.byte...
<== B123456789 * B987654321;;
==> B121932631112635269
<==

~B123456789;;
==> B~123456789
<==

B123456789 - B987654321;;
==> B~864197532
<==

B878675786879 + B868757575757;;
==> B1747433362636
<==

B12345678987654321 < B98765432123456789;;
==> true
<==

B98765432123456789 = B98765432123456789;;
==> true
<==

B4611686018427387904 + ~B4611686018427387904;;
==> B0
```