1. Denotational Semantics. Denotational semantics is a formal method to specify the meaning of programming languages. We define an abstract syntax for the language (*Term*), a meaning function (*Me*), a polymorphic store (*store*) and a polymorphic environment (*env*).

The code is laid out in this document in a logical form (rather than slavishly following the order the compiler wants), but we must force the `ML` code into the right order to make sure that things are defined before use.

⟨ Type Definitions 2 ⟩
⟨ Functions 52 ⟩
⟨ Meaning Function 6 ⟩;
⟨ Test Cases 54 ⟩

2. This is the type of possible declarations, along with the start of the definition of possible terms. They are mutually recursive definitions so they must be connected by an **and**. The actual declaration type are explained below in the definition of the semantic function for declarations.

⟨ Type Definitions 2 ⟩ ≡
datatype *Decl* =
Var_Decl **of string** × *Term*
| *Val_Decl* **of string** × *Term*
| *Rec_Decl* **of string** × *Term*
and ⟨ Term Definition 5 ⟩;

See also sections 47, 48, 49, 50, and 51.

This code is used in section 1.

3. *Term* **and** *Me***.** We define the datatype for the abstract syntax tree and the meaning function in parallel. `Web` will worry about putting it all together in the right order.

The meaning function has to have type:

$Me : Term \mapsto (Value\ \ env) \mapsto (Value\ \ continuation) \mapsto (Value\ \ store) \mapsto (Value \times (Value\ \ store))$.

That is, it maps abstract syntax trees, environments, continuations, and stores to a value,store pair.

4. The denotational semantics are given in terms of the following:

Me - this is the meaning function itself.
Ξ - is an invalid value
ε - an expression
ξ - a symbol
ν - a number
ρ - the environment (bindings from names to values)
θ - this is the continuation
Θ - this is a null continuation that simply returns the value
σ - this is the store of values
ϕ - a location in the store
$x[e/v]$ - a substitution of the expression e for the name (or whatever) v, where x is usually ρ or σ
$x[\![v]\!]$ - v in the context of x, where x is **Me**, ρ, σ, **O** or **V**
$\langle x, y\rangle$ - the tuple composed of x and y
$\{e\}$ - a continuation that evaluates e in the current context

When either of the environment or store are omitted, the environment or store from the enclosing environment is assumed.

5. The first type of term is the name of a variable or constant. The meaning of a name is the value that the environment contains for that name. For variables, this value is the location in the store where the value may be found.

$\langle$ Term Definition 5 $\rangle \equiv$
Term = *Var* **of string**

See also sections 7, 9, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, and 45.
This code is used in section 2.

6. $\mathbf{Me}[\![\xi]\!]\rho\theta\sigma = \theta\langle\rho[\![\xi]\!], \sigma\rangle$

$\langle$ Meaning Function 6 $\rangle \equiv$
fun *Me* (*Var x*) *e c s* = *c* (*lookup x e*, *s*)

See also sections 8, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, and 46.
This code is used in section 1.

7. Integer constants.

$\langle$ Term Definition 5 $\rangle+ \equiv$
| *Numeral* **of int**

8. $\mathbf{Me}[\![\nu]\!]\rho\theta\sigma = \theta\langle\mathbf{V}[\![\nu]\!], \sigma\rangle$

$\langle$ Meaning Function 6 $\rangle+ \equiv$
| *Me* (*Numeral n*) *e c s* = *c* (*intValue n*, *s*)

9. Declarations.

Declarations come in 3 flavours: `val`, `rec`, and `var`. For all three, the declaration only holds for the evaluation of the expression with which it is composed. `val` introduces a constant. `rec` is similar but the symbol is introduced into the environment of the expression (which must be a function) to allow for recursive calls. `var` introduces a variable.

⟨ Term Definition 5 ⟩+ ≡
  | *Decl* **of** *Decl* × *Term*

10. $\mathbf{Me}[\![\texttt{var}\ \xi = \varepsilon_1\ ;\ \varepsilon_2]\!]\rho\theta\sigma = \mathbf{Me}[\![\varepsilon_2]\!]\rho[\phi/\xi]\theta\sigma[\mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma/\phi]$

⟨ Meaning Function 6 ⟩+ ≡

```
| Me (Decl (Var_Decl (x, V), E)) e c s =
    Me V e (
          λ (R, _) ⇒ let
                 val (S, L) = new s R;
                 val nE = bind x (lvalueValue L) e;
               in
                 Me E nE c S
               end
        ) s
```

11. $\mathbf{Me}[\![\texttt{val}\ \xi = \varepsilon_1\ ;\ \varepsilon_2]\!]\rho\theta\sigma = \mathbf{Me}[\![\varepsilon_2]\!]\rho[\mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma/\xi]\theta\sigma$

⟨ Meaning Function 6 ⟩+ ≡

```
| Me (Decl (Val_Decl (x, V), E)) e c s =
    Me V e (
          λ (R, _) ⇒ Me E (bind x R e) c s
        ) s
```

12. The interpretation of the recursive declaration ($\mathbf{Me}[\![\texttt{rec}\ \xi = \varepsilon_1\ ;\ \varepsilon_2]\!]\rho\theta\sigma$) is fairly difficult to describe, short of translating the ML code. There are a couple of ways of doing this, but the one I chose is to implement the Y combinator (`fix`) in the language and then apply it to the functional we want to make recursive. To bootstrap the process, we put an entry for `fix` into the environment as a var name with an initial binding to an invalid value, which we replace once we have a definition for the function proper. If we were looking for efficiency, we'd put these in the initial environment and store.

⟨ Meaning Function 6 ⟩+ ≡

```
| Me (Decl (Rec_Decl (x, V), E)) e c s = let
      val fixpoint = Proc ("fix-f", Proc ("fix-x", App (App (Var "fix-f", App (Deref (Var "fix")
                         , Var "fix-f")), Var "fix-x")));
      val (S, L) = new s invalidValue;
      val nE = bind "fix" (lvalueValue L) e;
    in
      Me fixpoint nE (
            λ (FIX, _) ⇒
                Me (App (Deref (Var "fix"), Proc (x, V))) nE (
                      λ (funcValue R, S) ⇒ Me E (bind x (funcValue R) e) c S
                      | _ ⇒ raise NotFuncDecl
                    ) (update S L FIX)
          ) S
    end
```

13. Function operations.
Function abstraction. Define a function of one parameter bound in the scope of the expression.

$\langle$ Term Definition 5 $\rangle+\equiv$
  | *Proc* **of string**$\times$ *Term*

14. $\mathbf{Me}[\![\texttt{proc}\ \xi \texttt{=>} \varepsilon]\!]\rho\theta\sigma = \theta\langle\lambda\theta'.\lambda\langle V,\sigma'\rangle.\mathbf{Me}[\![\varepsilon]\!]\rho[V/\xi]\theta'\sigma',\sigma\rangle$

$\langle$ Meaning Function 6 $\rangle+\equiv$

```
| Me (Proc (x, E)) e c s =
    c (
      funcValue (
        λ C ⇒
          λ (V, S) ⇒ Me E (bind x V e) C S
      )
      , s
    )
```

15. Function Application. Supply one parameter to the function and execute the function.

$\langle$ Term Definition 5 $\rangle+\equiv$
  | *App* **of** *Term*$\times$ *Term*

16. The interpretation of function application ($\mathbf{Me}[\![\varepsilon_1(\varepsilon_2)]\!]\rho\theta\sigma$) is basically going to be a translation of the ML code.

$\langle$ Meaning Function 6 $\rangle+\equiv$

```
| Me (App (E1, E2)) e c s =
    Me E1 e (
        λ (funcValue f, _) ⇒
            Me E2 e (
                λ VS ⇒ f c VS
            ) s
        | _ ⇒ raise NotFunc
    ) s
```

17. Call the specified function passing our continuation as a functional parameter.

$\langle$ Term Definition 5 $\rangle+\equiv$
  | *Callcc* **of** *Term*

18. The interpretation of function application ($\mathbf{Me}[\![\texttt{callcc}\ \varepsilon]\!]\rho\theta\sigma$) is basically going to be a translation of the ML code.

$\langle$ Meaning Function 6 $\rangle+\equiv$

```
| Me (Callcc E) e c s =
    Me E e (
        λ (funcValue f, _) ⇒
            f c (
                funcValue (
                    λ (C : Value continuation) ⇒ c
                )
                , s
            )
        | _ ⇒ raise NotFunc
    ) s
```

19. Operations on Pairs.
Create a pair from the values of two expressions.

⟨ Term Definition 5 ⟩+ ≡
  | *Pair* **of** *Term* × *Term*

20. $\mathbf{Me}[\![<\varepsilon_1,\varepsilon_2>]\!]\rho\theta\sigma = \theta\langle\mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma, \mathbf{Me}[\![\varepsilon_2]\!]\rho\Theta\sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡
```
  | Me (Pair (E1, E2)) e c s =
      Me E1 e (
            λ (V1, _) ⇒
                Me E2 e (
                      λ (V2, S) ⇒ c (pairValue (V1, V2), s)
                   ) s
         ) s
```

21. Get the first element from a pair.

⟨ Term Definition 5 ⟩+ ≡
  | *Fst* **of** *Term*

22. $\mathbf{Me}[\![\texttt{fst}\ \varepsilon]\!]\rho\theta\sigma = \theta\langle\pi_1(\mathbf{Me}[\![\varepsilon]\!]\rho\Theta\sigma), \sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡
```
  | Me (Fst T) e c s =
      Me T e (
            λ (pairValue (T1, T2), _) ⇒ c (T1, s)
             | _ ⇒ raise NotPair
         ) s
```

23. Get the second element from a pair.

⟨ Term Definition 5 ⟩+ ≡
  | *Snd* **of** *Term*

24. $\mathbf{Me}[\![\texttt{snd}\ \varepsilon]\!]\rho\theta\sigma = \theta\langle\pi_2(\mathbf{Me}[\![\varepsilon]\!]\rho\Theta\sigma), \sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡
```
  | Me (Snd T) e c s =
      Me T e (
            λ (pairValue (T1, T2), _) ⇒ c (T2, s)
             | _ ⇒ raise NotPair
         ) s
```

25. Flow of control.
Conditional execution. Only one of the then or else expressions will be executed.

⟨Term Definition 5⟩+ ≡
  | *Cond* **of** *Term* × *Term* × *Term*

26. $\mathbf{Me}[\![\texttt{if}\ \varepsilon_1\ \texttt{then}\ \varepsilon_2\ \texttt{else}\ \varepsilon_3\ \texttt{fi}]\!]\rho\theta\sigma = (\mathit{if}\ \mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma = \texttt{true}\ \mathit{then}\ \mathbf{Me}[\![\varepsilon_2]\!]\ \mathit{else}\ \mathbf{Me}[\![\varepsilon_3]\!])\rho\theta\sigma$

⟨Meaning Function 6⟩+ ≡
```
  | Me (Cond (E1, E2, E3)) e c s =
      Me E1 e (
            λ (boolValue true, _) ⇒ Me E2 e c s
             | (boolValue false, _) ⇒ Me E3 e c s
             | _ ⇒ raise NotBool
          ) s
```

27. Expression composition. Execute the expressions sequentially.

⟨Term Definition 5⟩+ ≡
  | *Seq* **of** *Term* × *Term*

28. $\mathbf{Me}[\![\varepsilon_1;\varepsilon_2]\!]\rho\theta\sigma = \mathbf{Me}[\![\varepsilon_1]\!]\rho\{\mathbf{Me}[\![\varepsilon_2]\!]\rho\theta\}\sigma$

⟨Meaning Function 6⟩+ ≡
```
  | Me (Seq (E1, E2)) e c s =
      Me E1 e (
            λ (V, S) ⇒ Me E2 e c S
          ) s
```

29. Indefinite iteration. Execute the second expression as long as the first expression evaluates to `true`.

⟨Term Definition 5⟩+ ≡
  | *While* **of** *Term* × *Term*

30. $\mathbf{Me}[\![\texttt{while}\ \varepsilon_1\ \texttt{do}\ \varepsilon_2\ \texttt{od}]\!]\rho\theta\sigma = (\mathit{if}\ \mathbf{Me}[\![\varepsilon_1]\!]\Theta\sigma = \texttt{true}\ \mathit{then}\ \mathbf{Me}[\![\varepsilon_2]\!]\rho\{\mathbf{Me}[\![\texttt{while}\ldots]\!]\rho\theta\}\ \mathit{else}\ \theta)\sigma$

⟨Meaning Function 6⟩+ ≡
```
  | Me (While (E1, E2)) e c s =
      Me E1 e (
            λ (boolValue true, S) ⇒
                Me E2 e (
                      λ (V, S) ⇒ Me (While (E1, E2)) e c S
                    ) S
             | (boolValue false, S) ⇒ c (invalidValue, S)
             | _ ⇒ raise NotBool
          ) s
```

31. Storage References and Updates.
Allocate storage for a value and return the location of the value in the store.

$\langle$ Term Definition 5 $\rangle + \equiv$
  | *Ref* **of** *Term*

32. $\mathbf{Me}[\![\texttt{ref}\ \varepsilon]\!]\rho\theta\sigma = \theta\langle\phi, \sigma[\mathbf{Me}[\![\varepsilon]\!]\rho\Theta\sigma/\phi]\rangle$

$\langle$ Meaning Function 6 $\rangle + \equiv$

```
| Me (Ref E) e c s =
    Me E e (
        λ (V, _) ⇒ let
                val (nS, L) = new s V;
            in
                c (lvalueValue L, nS)
            end
    ) s
```

33. Given the location of a datum, get the value currently stored there.

$\langle$ Term Definition 5 $\rangle + \equiv$
  | *Deref* **of** *Term*

34. $\mathbf{Me}[\![.\varepsilon]\!]\rho\theta\sigma = \theta\langle\sigma[\![\mathbf{Me}[\![\varepsilon]\!]\rho\Theta\sigma]\!], \sigma\rangle$

$\langle$ Meaning Function 6 $\rangle + \equiv$

```
| Me (Deref E) e c s =
    Me E e (
        λ (lvalueValue V, S) ⇒ c (access S V, s)
         | _ ⇒ raise NotLValue
    ) s
```

35. Modify the storage at the location specified by the first expression to have the value of the second expression.

$\langle$ Term Definition 5 $\rangle + \equiv$
  | *Assign* **of** *Term* × *Term*

36. $\mathbf{Me}[\![\varepsilon_1{:=}\varepsilon_2]\!]\rho\theta\sigma = \theta\langle\mathbf{Me}[\![\varepsilon_2]\!]\rho\Theta\sigma, \sigma[\mathbf{Me}[\![\varepsilon_2]\!]\rho\Theta\sigma/\mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma]\rangle$

$\langle$ Meaning Function 6 $\rangle + \equiv$

```
| Me (Assign (E1, E2)) e c s =
    Me E1 e (
        λ (lvalueValue L, _) ⇒
            Me E2 e (
                λ (V, _) ⇒ c (V, update s L V)
            ) s
         | _ ⇒ raise NotLValue
    ) s
```

37. Integer Operations.
Add two integers together.

⟨ Term Definition 5 ⟩+ ≡
| *Add* **of** *Term* × *Term*

38. $\mathbf{Me}[\![\varepsilon_1{+}\varepsilon_2]\!]\rho\theta\sigma = \theta\langle\mathbf{O}[\![+]\!](\mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma, \mathbf{Me}[\![\varepsilon_2]\!]\rho\Theta\sigma), \sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡
```
| Me (Add (E1, E2)) e c s =
    Me E1 e (
        λ (intValue V1, _) ⇒
            Me E2 e (
                λ (intValue V2, _) ⇒ c (intValue (V1 + V2), s)
                 | _ ⇒ raise NotInteger
            ) s
         | _ ⇒ raise NotInteger
    ) s
```

39. Multiply two integers.

⟨ Term Definition 5 ⟩+ ≡
| *Mult* **of** *Term* × *Term*

40. $\mathbf{Me}[\![\varepsilon_1{*}\varepsilon_2]\!]\rho\theta\sigma = \theta\langle\mathbf{O}[\![\times]\!](\mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma, \mathbf{Me}[\![\varepsilon_2]\!]\rho\Theta\sigma), \sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡
```
| Me (Mult (E1, E2)) e c s =
    Me E1 e (
        λ (intValue V1, _) ⇒
            Me E2 e (
                λ (intValue V2, _) ⇒ c (intValue (V1*V2), s)
                 | _ ⇒ raise NotInteger
            ) s
         | _ ⇒ raise NotInteger
    ) s
```

41. Calculate the negative of an integer.

⟨ Term Definition 5 ⟩+ ≡
| *Neg* **of** *Term*

42. $\mathbf{Me}[\![-\varepsilon]\!]\rho\theta\sigma = \theta\langle -\mathbf{Me}[\![\varepsilon]\!]\rho\Theta\sigma, \sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡
```
| Me (Neg E) e c s =
    Me E e (
        λ (intValue V, _) ⇒ c (intValue (0 − V), s)
         | _ ⇒ raise NotInteger
    ) s
```

43. Boolean Operations. Determine if the first integer expression is lower in value than the second. Return a boolean truth value.

⟨ Term Definition 5 ⟩+ ≡
  | *Less* **of** *Term* × *Term*

44. $\mathbf{Me}[\![\varepsilon_1 {<} \varepsilon_2]\!]\rho\theta\sigma = \theta\langle \mathbf{O}[\![<]\!](\mathbf{Me}[\![\varepsilon_1]\!]\rho\Theta\sigma, \mathbf{Me}[\![\varepsilon_2]\!]\rho\Theta\sigma), \sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡

```
  | Me (Less (E1, E2)) e c s =
      Me E1 e (
            λ (intValue V1, _) ⇒
                Me E2 e (
                      λ (intValue V2, _) ⇒ c (boolValue (V1 < V2), s)
                      | _ ⇒ raise NotInteger
                ) s
            | _ ⇒ raise NotInteger
      ) s
```

45. Calculate the boolean complement of the expression.

⟨ Term Definition 5 ⟩+ ≡
  | *Not* **of** *Term*

46. $\mathbf{Me}[\![\neg\varepsilon]\!]\rho\theta\sigma = \theta\langle \neg\mathbf{Me}[\![\varepsilon]\!]\rho\Theta\sigma, \sigma\rangle$

⟨ Meaning Function 6 ⟩+ ≡

```
  | Me (Not E) e c s =
      Me E e (
            λ (boolValue true, _) ⇒ c (boolValue false, s)
            | (boolValue false, _) ⇒ c (boolValue true, s)
            | _ ⇒ raise NotBool
      ) s
```

47. Environments. An environment captures the bindings of names to values (in this case locations). This trivial implementation has dreadful performance, but it is at least fairly obviously correct.

⟨ Type Definitions 2 ⟩+ ≡

```
abstype α env =
    Env of string ↦ α
with
  exception unbound_variable of string;
  val newenv =
    Env (
          λ x ⇒ raise unbound_variable x
        );
  fun bind x t (Env f) =
      Env (
            λ y ⇒
                if x = y then
                  t
                else
                  f y
          );
  fun lookup x (Env f) = f x
end;
```

48. Stores. A store binds locations to values. Once again, this is just about the most inefficient implementation you could come up with (although there actually was a bug in the first version). A store can be thought of as a mapping from locations to values, where new locations are created on demand.

⟨ Type Definitions 2 ⟩+ ≡

```
abstype α store =
      Store of int×(int ↦ α)
   and lvalue =
      lvalue of int
with
   exception segmentation_violation;
   val newstore =
     Store (
         0
         , λ x ⇒ raise segmentation_violation
        );
   fun new (Store (avail, f)) v =
       ( Store (
              avail + 1
              , λ l ⇒
                     if l = avail then
                        v
                     else
                        f l
             )
          , lvalue (avail)
         );
   fun access (Store (_, f)) (lvalue loc) = f loc;
   fun update (Store (avail, f)) (lvalue loc) v =
       Store (
            avail
            , λ l ⇒
                   if l = loc then
                      v
                   else
                      f l
           );
end;
```

49. Values and Miscellaneous Definitions. Define a continuation to be a mapping from a value,state pair to a resulting value,state.

⟨ Type Definitions 2 ⟩+ ≡
  type α *continuation* $= (\alpha \times \alpha\ store) \mapsto (\alpha \times \alpha\ store)$;

50. Values. These are the set of values that can result from a computation. Only the first 3 of these were in the original problem description. I added the others to make the language/interpreter more useful and to catch and report errors.

⟨ Type Definitions 2 ⟩+ ≡
  datatype *Value* =
      intValue **of int**
    | *lvalueValue* **of** *lvalue*
    | *funcValue* **of** *Value continuation* $\mapsto$ *Value continuation*
    | *boolValue* **of** *bool*
    | *pairValue* **of** *Value* $\times$ *Value*
    | *invalidValue*
    | *Missing_Name* **of string**;

51. These are exceptions that are raised by the semantic function when values are inappropriate.

⟨ Type Definitions 2 ⟩+ ≡
  exception *NotImplemented*;
  exception *NotCorrect*;
  exception *NotPair*;
  exception *NotLValue*;
  exception *NotBool*;
  exception *NotInteger*;
  exception *NotFunc*;
  exception *NotFuncDecl*;

52. Here is an `ML` version of the Y fix point operator. We could use it for to implement recursion. It would only be a minor change to make the semantic function use this rather than use `ML`'s builtin recursion.

⟨ Functions 52 ⟩ ≡
  fun *fix f x* = *f* (*fix f*) *x*;

See also section 53.

This code is used in section 1.

53. The empty continuation.

⟨ Functions 52 ⟩+ ≡
  fun *nullContinuation x* = *x*;

54. Test of the Machine. Testing is no substitute for correct implementation, but it is somewhat heartwarming to see such an abstract interpreter actually work.

First of all define a *test* function that will accept any program in the language and execute it, returning the result.

⟨ Test Cases 54 ⟩ ≡
```
fun test n = let
      val (V, S) =
        Me n newenv nullContinuation newstore
            handle unbound_variable x ⇒ (Missing_Name x, newstore);
    in
      V
    end;
```
See also sections 55, 56, 57, 58, and 59.
This code is used in section 1.

55. A few dead simple tests to show that integers and pairs work properly.

⟨ Test Cases 54 ⟩+ ≡
```
test (Numeral 3);
test (Neg (Add (Numeral 3, Numeral 39)));
test (Fst (Pair (Numeral 3, Numeral 4)));
test (Snd (Pair (Numeral 3, Numeral 4)));
test (Seq (Numeral 3, Numeral 4));
```

56. Test binding names, both through value declarations and λ bindings. Also verify that function abstraction, function application, and call-with-current-continuation work.

⟨ Test Cases 54 ⟩+ ≡
```
val X = "x";
val Y = "y";
val FIX = "fix"
and F = "f";
test (Decl (Val_Decl (X, Numeral 29), Var X));
test (App (Proc (X, Numeral 17), Numeral 7));
test (App (Proc (X, Add (Var X, Numeral 1)), Numeral 7));
test (Decl (Val_Decl (X, Proc (Y, Var Y)), Add (Numeral 3, App (Var X, Numeral 55))));
test (Decl (Val_Decl (X, Proc (Y, Seq (App (Var Y, Numeral 33), Numeral 44)))
            , Add (Numeral 3, Callcc (Var X))));
```

57. Now for some real excitement: recursive functions. Surprise! Surprise! They actually work.

⟨ Test Cases 54 ⟩+ ≡
```
val fact = Rec_Decl (X, Proc (Y, Cond (Less (Var Y, Numeral 2)
                       , Numeral 1, Mult (App (Var X, Add (Var Y, Neg (Numeral 1))), Var Y))));
test (Decl (fact, App (Var X, Numeral 1)));
test (Decl (fact, App (Var X, Numeral 5)));
```

58. Ok, so let's use some CPU time! Try fibonacci

⟨ Test Cases 54 ⟩+ ≡

val *fib* = *Rec_Decl* (*X*, *Proc* (*Y*, *Cond* (*Less* (*Var Y*, *Numeral* 2), *Numeral* 1, *Add* (*App* (*Var X*, *Add* (*Var Y*, *Neg* (*Numeral* 1))), *App* (*Var X*, *Add* (*Var Y*, *Neg* (*Numeral* 2)))))));
test (*Decl* (*fib*, *App* (*Var X*, *Numeral* 1)));
test (*Decl* (*fib*, *App* (*Var X*, *Numeral* 5)));
test (*Decl* (*fib*, *App* (*Var X*, *Numeral* 10)));
test (*Decl* (*fib*, *App* (*Var X*, *Numeral* 15)));
test (*Decl* (*fib*, *App* (*Var X*, *Numeral* 20)));
test (*Decl* (*fib*, *App* (*Var X*, *Numeral* 25)));
test (*Decl* (*fib*, *App* (*Var X*, *Numeral* 28)));

59. Finally test that reference bindings and iteration work properly.

⟨ Test Cases 54 ⟩+ ≡

test (*Assign* (*Ref* (*Numeral* 2), *Numeral* 3));
test (*Decl* (*Var_Decl* (*X*, *Numeral* 1), *Deref* (*Var X*)));
test (*Decl* (*Var_Decl* (*X*, *Numeral* 1), *Seq* (*Assign* (*Var X*, *Add* (*Numeral* 22, *Deref* (*Var X*))), *Deref* (*Var X*))));
test (*Decl* (*Var_Decl* (*X*, *Numeral* 1), (*Decl* (*Var_Decl* (*Y*, *Numeral* 0), *Seq* (*While* (*Less* (*Deref* (*Var X*), *Numeral* 11), *Seq* (*Assign* (*Var Y*, *Add* (*Deref* (*Var Y*), *Deref* (*Var X*))), (*Assign* (*Var X*, *Add* (*Numeral* 1, *Deref* (*Var X*)))))), *Deref* (*Var Y*))))));

60. Index.

access: 34, 48.
Add: 37, 38, 55, 56, 57, 58, 59.
App: 12, 15, 16, 56, 57, 58.
Assign: 35, 36, 59.
avail: 48.
bind: 10, 11, 12, 14, 47.
bool: 50.
boolValue: 26, 30, 44, 46, 50.
Callcc: 17, 18, 56.
Cond: 25, 26, 57, 58.
continuation: 3, 18, 49, 50.
Decl: 2, 9, 10, 11, 12, 56, 57, 58, 59.
Deref: 12, 33, 34, 59.
env: 1, 3, 47.
Env: 47.
E1: 16, 20, 26, 28, 30, 36, 38, 40, 44.
E2: 16, 20, 26, 28, 30, 36, 38, 40, 44.
E3: 26.
fact: 57.
fib: 58.
fix: 52.
FIX: 12, 56.
fixpoint: 12.
Fst: 21, 22, 55.
funcValue: 12, 14, 16, 18, 50.
intValue: 8, 38, 40, 42, 44, 50.
invalidValue: 12, 30, 50.
Less: 43, 44, 57, 58, 59.
loc: 48.
lookup: 6, 47.
lvalue: 48, 50.
lvalueValue: 10, 12, 32, 34, 36, 50.
Me: 1, 3, 6, 8, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 54.
Missing_Name: 50, 54.
Mult: 39, 40, 57.
nE: 10, 12.
Neg: 41, 42, 55, 57, 58.
new: 10, 12, 32, 48.
newenv: 47, 54.
newstore: 48, 54.
Not: 45, 46.
NotBool: 26, 30, 46, 51.
NotCorrect: 51.
NotFunc: 16, 18, 51.
NotFuncDecl: 12, 51.
NotImplemented: 51.
NotInteger: 38, 40, 42, 44, 51.
NotLValue: 34, 36, 51.
NotPair: 22, 24, 51.
nS: 32.
nullContinuation: 53, 54.
Numeral: 7, 8, 55, 56, 57, 58, 59.
Pair: 19, 20, 55.
pairValue: 20, 22, 24, 50.
Proc: 12, 13, 14, 56, 57, 58.
Rec_Decl: 2, 12, 57, 58.
Ref: 31, 32, 59.
segmentation_violation: 48.
Seq: 27, 28, 55, 56, 59.
Snd: 23, 24, 55.
store: 1, 3, 48, 49.
Store: 48.
Term: 1, 2, 3, 5, 9, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45.
test: 54, 55, 56, 57, 58, 59.
T1: 22, 24.
T2: 22, 24.
unbound_variable: 47, 54.
update: 12, 36, 48.
Val_Decl: 2, 11, 56.
Value: 3, 18, 50.
Var: 5, 6, 12, 56, 57, 58, 59.
Var_Decl: 2, 10, 59.
VS: 16.
V1: 20, 38, 40, 44.
V2: 20, 38, 40, 44.
While: 29, 30, 59.

⟨ Functions 52, 53 ⟩ Used in section 1.
⟨ Meaning Function 6, 8, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46 ⟩ Used in section 1.
⟨ Term Definition 5, 7, 9, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45 ⟩ Used in section 2.
⟨ Test Cases 54, 55, 56, 57, 58, 59 ⟩ Used in section 1.
⟨ Type Definitions 2, 47, 48, 49, 50, 51 ⟩ Used in section 1.

Denotational Semantics and an ML Interpreter for a Functional Programming Language

for CS742, Dominic Duggan

Table of Contents

	Section	Page
Denotational Semantics	1	1
Term and *Me*	3	2
Declarations	9	3
Function operations	13	4
Operations on Pairs	19	5
Flow of control	25	6
Storage References and Updates	31	7
Integer Operations	37	8
Boolean Operations	43	9
Environments	47	10
Stores	48	11
Values and Miscellaneous Definitions	49	12
Test of the Machine	54	13
Index	60	15