Description of a Very Small Functional Programming Language

Purpose

This language is intended as a programming project for CSCI 5220, Program Translation. The project is to write a compiler for this language.

Introduction

Functional programming languages have been around since the early days of Lisp. Based on Church's lambda-calculus, they have a few common characteristics.

1. All computation is performed by using pure functions. A function takes a parameter and produces a result. There is no notion of a side effect. There are no variables, or anything that can change while a program is computing. Data structures cannot be modified.

2. Programs are declarative. They consist of a collection of facts.

3. Functions are considered values, just like other values. You can pass a function to another function, receive a function as the result of a function, etc.

4. Implementation issues such as memory management the pollute the pure notion of functions are handled by the language implementation, not by the programmer.

To somebody who has only used imperative programming languages, functional languages initially seem very strange, to the point of being impossible to use. How can you compute if you cannot change anything? How can you manage without variables? And yet, functional languages are popular among programmers who bother to learn how to use them. The reason is that they tend to yield short, simple programs for some problems that would be quite difficult to solve in other styles. Programmers often find that their programs work the first time they are tried, or with only a tiny amount of fixing, far more often than even the programmers feel they have a right to expect.

Examples of functional programming languages are Lisp, Standard ML and Haskell. There are many others. Not all functional languages offer *only* functional programming. Standard ML, for example, also offers imperative features. We are only interested here in a pure language, though.

A relatively recent addition to the family is Mondrian, which is a bare-bones functional language designed to be implemented in a nontraditional way, using an object-oriented abstract machine (Microsoft's .NET machine). It is possible to write function definitions in Mondrian and to call those functions from programs written in other programming languages, such as C#.

This document describes a functional programming language that is even more bare-bones than Mondrian. We will call it Minnie. The syntax of Minnie is a hybrid of Haskell, Mondrian and other languages, and is intended to be very simple.

Values and types

The values of Minnie have the following forms. Each value has exactly one type.

true, **false**

Values **true** and **false** are simple values used as the results of tests. These values have type **bool**.

Integers

Integers (..., -2, -1, 0, 1, 2, ...) are values. There is no limit in the language definition on the size of an integer. Integers have type **int**.

'a', 'b', ...

Characters 'a', 'b', etc. are values. The alphabet is the ASCII alphabet. Characters have type **char**.

[x,y,z]

A list of zero or more values *of the same type* is a value. We will write lists in square brackets, with commas separating the list members. For example, [1,2,3] is a list of three integers. [3] is a list of integers that has just one integer in it. ['a','b'] is a list of two characters. [] is the empty list. A list whose members have type *T* has type **list *T***.

(x,y)

Ordered pairs such as (3,4) are values. An ordered pair contains exactly two values, not required to be of the same type. For example, (5,'a') is an ordered pair holding an integer and a character. The order matters. We write ordered pairs in parentheses, with a comma separating the parts.

nonnull(x) and **null**

For every type *T* there is a type **maybe** *T*. A member of this either has the form nonnull(*x*) (indicating value *x*, of type *T*) or the special value **null**, indicating no value. This is used in cases where you *might* have a value of type *T*, and you might not.

functions

Functions are values. (For technical reasons, not every possible function is a value, but that is of little relevance for this description.) Each function takes exactly one parameter, and produces exactly one result. A function that takes a parameter of type *A* and produces a result of type *B* has type *A* -> *B*. Type *A* is called the *domain* of the function, and type *B* is called the *codomain* of the function.

states

There is a type called **state**. The state type is discussed below.

Names

An *name*, or an *identifier*, is a sequence of letters containing at least one letter. The reserved words **and**, **bool**, **case**, **char**, **def**, **else**, **end**, **evaluate**, **false**, **in**, **int**, **let**, **list**, **maybe**, **not**, **or**, **return**, **state**, **true** and **type** are not allowed to be used as identifiers.

Identifiers are used to name values. They are not variables. They just refer to things.

Expressions

An expression is evaluated to produce a value. The expressions have the following forms.

x

> An identifier is an expression, assuming it occurs in a context where that identifier is defined. It stands for the value that the identifier names. So *x* is the thing named by *x*.

A B

> If *A* and *B* are expressions, then *A B* is an expression. The value of *A* must be a function. If the value of *A* is function *f*, and the value of *B* is value *v*, then evaluating *A B* produces the value *f* (*v*), the result of applying function *f* to value *v*.

case C_1 => E_1 | ... **else** => E_{n+1} **end**

> Expression **case** C_1 => E_1 | C_2 => E_2 | ... C_n=> E_n | **else** => E_{n+1} **end** is evaluated by testing conditions C_1, ..., C_n in the order written. If expression C_1 evaluates to **true**, then the value of this **case** expression is the value of expression E_1. If C_1 evaluates to **false**, but C_2 evaluates to **true**, then the value of expression E_2 is the value of the case expression. In general the value is the value of E_i where C_i is the first of the conditions whose value is **true**. If none of the conditions is true, then the value is E_{n+1}.
>
> The **else** phrase is required. There must be at least one condition. So $n > 0$.

let *x* = *A* **in** *B* **end**

> If *x* is an identifier and *A* and *B* are expressions then expression **let** *x* = *A* **in** *B* **end** has the value of expression *B*, but while *B* is evaluated, identifier *x* names the value of expression *A*. Expression **let** *x* = *A* **in** *B* **end** has the same value as ((\ *x* -> *B*) *A*).

\ *x* -> *A*

> The value of expression \ *x* -> *A* is a function *f* defined by *f* (*x*) = *A*. What occurs after the \ can be an identifier, and that identifier can be used in expression *A*.

(*A*)

> Expression (*A*) has the same meaning as expression *A*.

[*A*,*B*,*C*]

> Any sequence of expression separated by commas and surrounded by [...] is an expression whose value

is a list of the values of the expressions listed. There can be any number of expressions in the list, including none. For example, expression [2+3, 8+4] has value [5,12]. Expression [*A*,*B*,*C*] is has the same meaning as ((*A*):(*B*):(*C*):nil).

(*A*,*B*)

If expression *A* has value *u* and expression *B* has value *v*, then expression (*A*,*B*) has value (*u*,*v*), an ordered pair.

(*A*,*B*,*C*)
(*A*,*B*,*C*,*D*)

(*A*,*B*,*C*) abbreviates (*A*,(*B*,*C*)). (*A*,*B*,*C*,*D*) abbreviates (*A*,(*B*,(*C*,*D*))). In general, any positive number of expressions can occur in parentheses, separated by commas. The comma is thought of as associating to the right.

"abc"

"abc" abbreviates ['a', 'b', 'c']. The characters can include \n (for newline) and \" (for a quot) in string constants.

'a'

'a' is the character constant (lower case a). Character constants allow letters, digits, special characters (other than the newline character) and the special sequence '\n' (a newline character).

1234

Integer constants are written in standard decimal notation.

A == *B*

Expresson *A* == *B* has value **true** if *A* and *B* have the same value, and **false** if *A* and *B* have different values.

A < *B*

Expression *A* < *B* has value **true** if the value of *A* is strictly less than the value of *B*, and **false** otherwise.

A **or** *B*

A **or** *B* is equivalent to **case** *A* => **true** | **else** => *B* **end**.

A **and** *B*

A **and** *B* is equivalent to **case** *A* => *B* | **else** => **false end**.

not *A*

not *A* is equivalent to **case** *A* => **false** | **else** => **true end**.

A + *B*

A + *B* produces the sum of the values of *A* and *B*.

A – *B*

A – *B* produces the difference of the values of *A* and *B*.

A * *B*

A * *B* produces the product of the values of *A* and *B*.

A / *B*

A / *B* produces the integer quotient of the values of *A* and *B*. For example, expresssion 5/3 has value 1.

A : *B*

A : *B* produces a list *L* such that head(*L*) is the value of expression *A*, and tail(*L*) is the value of expression *B*.

A ++ *B*

A ++ *B* produces the concatenation of lists *A* and *B* (or really, of the lists that are the values of expression *A* and *B*).

A ; *B*

See monads.

return *A*

See monads.

(*x* <– *A*) ; *B*

See monads.

Precedence

Ambiguity is resolved by the following precedence and associativity rules. The table lists operators from high precedence to low precedence, with an associativity for each.

(juxtaposition)	left
not, **return**	n/a
*, /	left
+, –	left
:	right
++	right
==, <	none (*x* == *y* == *z* is not allowed)

and	left
or	left
\	n/a
;	left

Types and polymorphism

Every expression has a type. However, the type can by *polymorphic*, indicating that it can actually have many types. Polymorphism is stated in terms of type variables, where a type variable stands for an arbitrary type. Minnie uses identifiers for type variables.

(Polymorphic) types are described by *type expressions* whose values are (polymorphic) types. The following forms of type expression can be used.

bool, **int**, **char**, **state**

A type name stands for a basic type.

list *T*

Type **list** *T* is the type of lists whose members have type *T*.

maybe *T*

Type **maybe** *T* is the type of nonnull(*v*) and of the special value **null**.

(*A*,*B*)

Type (*A*,*B*) is the type of an ordered pair whose left-hand component has type *A* and whose right-hand component has type *B*.

(*A*,*B*,*C*)
(*A*,*B*,*C*,*D*)

(*A*,*B*,*C*) abbreviates (*A*,(*B*,*C*)). (*A*,*B*,*C*,*D*) abbreviates (*A*,(*B*,(*C*,*D*))). You can have any number of parts, with the comma associating to the right.

A -> *B*

A function whose domain is type *A* and whose codomain is type *B* has type *A* -> *B*.

(*A*)

Type expression (*A*) means the same thing as *A*.

x

An identifier is a type variable, standing for an arbitrary type. You can use any identifier for a type variable.

Precedence and associativity for type expressions

The -> operator has lower precedence than the unary **list** and **maybe** operators, and associates to the right.

Copying polymorphic types

Sometimes you need to copy a polymorphic type. A copy is made by choosing new names for the variables. For example, a copy of type (a,b) -> a might be (x,y) -> x.

Type rules for expressions

The following rules indicate how type checking is to be performed.

x

The type of an identifier depends on how the identifier was defined. If defined in a declaration, then the type is a *copy* of the type of x in its definition. If x is defined in a **let** expression, then the type of each use of x is a *copy* of the type under which it is defined in the let. If x is a function parameter, then the type of each use is exactly the same as it is in the function definition.

A B

If A B has type S and B has type T, then A has type S -> T.

case C_1 => E_1 | ... **else** => E_{n+1} **end**

Each condition C_i has type **bool**. All of the values E_i must have the same type.

let $x = A$ **in** B **end**

The type of each occurrence of x in expression B is a copy of the type of A. The **let** expression has the same type as B.

\ x -> A

The type of this expression is some function type S -> T. Each occurrence of x in A has type S. The type of A is T.

(A)

The type of (A) is the same as the type of A.

$[A,B,C]$

All of A, B and C have the same type, say T. The list has type **list** T. The same idea extends to arbitrary list expressions, with zero or more members.

(A,B)

If *A* has type *S* and *B* has type *T* then this pair expression has type (*S*,*T*).

"abc"

A string has type **list char**.

'a'

A character constant has type **char**.

1234

An integer constant has type **int**.

A == *B*

Expressions *A* and *B* must have type **int**. The equality test expression itself has type **bool**.

A < *B*

Expressions *A* and *B* must have type **int**. The comparison expression itself has type **bool**.

A **or** *B*

Expressions *A* and *B* must have type **bool**. The **or** expression itself has type **bool**.

A **and** *B*

Expressions *A* and *B* must have type **bool**. The **and** expression itself has type **bool**.

not *A*

Expressions *A* must have type **bool**. The **not** expression itself has type **bool**.

A + *B*

A and *B* have type **int**. Expression *A* + *B* also has type **int**.

A – *B*

A and *B* have type **int**. Expression *A* - *B* also has type **int**.

A * *B*

A and *B* have type **int**. Expression *A* * *B* also has type **int**.

A / *B*

A and *B* have type **int**. Expression *A* / *B* also has type **int**.

A : *B*

A has a type *T*, *B* type **list** *T*, and expression *A* : *B* has type **list** *T*.

A ++ *B*

A has a type **list** *T*, *B* type **list** *T*, and expression *A* : *B* has type **list** *T*.

A ; *B*

A has type **state** -> **maybe state**. *B* has type **state** -> **maybe**(**state**,*T*). Expression *A* ; *B* has type **maybe**(**state**,*T*).

return *A*

If *A* has type *T* then **return** *A* has type **state** -> **maybe**(**state**,*T*).

(*x* <- *A*) ; *B*

See monads.

Standard functions and constants

The following definitions are available to all programs.

Name	Type	Meaning
nil	**list** x	nil is the empty list.
head	list x -> x	Produce the head (the first member) of a list.
tail	**list** x -> **list** x	Produce the tail (all but the first member) of a list.
nilq	**list** *x* -> **bool**	nilq(*v*) is true just when *v* is an empty list.
left	(*x*,*y*) -> *x*	Produce the left-hand member of an ordered pair.
right	(*x*,*y*) -> *y*	Produce the right-hand member of an ordered pair.
null	**maybe** *x*	This is the null value.
nonnull	*x* -> **maybe** *x*	Produce a nonnull value.
nullq	**maybe** *x* -> **bool**	nullq(*v*) is true just when *v* is null.
value	**maybe** *x* -> *x*	Get the value inside a nonnull **maybe** item.
readInt	**state** -> **maybe**(**state**,**int**)	Read an integer and modify the state. See monads
readChar	**state** -> **maybe**(**state**,**char**)	Read a character and modify the state. See monads
write	*x* -> **state** -> **maybe state**	Write a string that describes *x* to the standard output. See monads

seq	(**state** -> **maybe state**) -> (**state** -> **maybe** (**state**, *x*)) -> (**state** -> **maybe** (**state**, *x*))	See monads.
bindseq	(**state** -> **maybe** (**state**, *x*)) -> (*x* -> **state** -> **maybe** (**state**, *y*)) -> (**state** -> **maybe** (**state**, *y*))	See monads.

Error handling

Sometimes a function cannot produce a value. For example, you cannot divide by 0, and you cannot take the head of an empty list. All such errors will cause a program to stop immediately. There is no support for recovering from sending bad data to one of the standard functions or operators.

Programs and declarations

A program consists of a sequence of declarations. The following forms of declaration can be used.

type *x*::*T* **end**

This indicates that identifier *x* will be defined later with polymorphic type *T*.

type *x*::*T* **def** *x* = *A* **end**

This is used to define identifier *x* to be the value of expression *A*. The type of *x* and *A* must be *T*.

type *x*::*T* **def** *x y* = *A* **end**

This abbreviates **type** *x*::*T* **def** *x* = \ *y* -> *A***end**.

type *x*::*T* **def** *x y z* = *A* **end**

This abbreviates **type** *x*::*T* **def** *x* = \ *y* -> (\z -> *A*) **end**. This idea extends to any number of function parameters.

evaluate *A* **end**

Create a state *S* = (*I*,[]), where *I* is the contents of the standard input. Compute the value of expression (*A S*), which must have type **maybe state**. If the result is nonnull(*s*') where state *s*' is (-,*str*), then print string str. If the result is null, then print an error message (evaluation failed).

Expression evaluation: Strict vs. nonstrict languages

Functional languages can be either *strict* or *nonstrict*.

A strict language is also called an eager language, or a call-by-value language. In a strict language, you evaluate the parameters of a function before you run the function.

Standard ML is an example of a strict language.

A nonstrict language is also called a lazy, or call-by-need language. In a nonstrict language, you evaluate the parameters of a function only when their values are needed (by primitive functions). Mondrian is an example of a nonstrict language.

Minnie should be nonstrict, at least partially. Each function should return immediately when it is called, without performing any evaluation. The evaluation should be done when the result of the function call is needed.

Monads and states

Programmers are typically used to the idea of imperative programming, where there is a state that is manipulated by the program. The state can contain values of variables, contents of files, etc.; everything that the program can modify is part of the state.

Manipulation of a state can be simulated in a functional language by explicitly passing the state as a parameter to each function. Each function must, additionally, produce a new state as part of its result. The state produced by one function is sent to the next one, and the states are threaded through the function calls. In the end, you can look at the state in the result to find out what a function did to the state.

Although you can write states explicitly, doing so can be unpleasant. Monads are a way of making that more pleasant. Monads also offer a convenient method of handling exceptions without introducing exception handling into the programming language. There is a pure, mathematical definition of a monad, but most programming languages provide a slightly modified form.

Minnie supports monads. In general, the state could be anything. In Minnie, as state is very simple and rigid. It contains two strings (*I*,*O*)where *I* is the unread part of the standard input, and *O* is what has been written to the standard output. The readInt, readChar and write functions are designed to manipulate the state. For example, readChar(state("abcd", "xyz")) = nonnull(state("bcd", "xyz"), 'a'). Notice that readChar indicates a change in the standard input, but not in the standard output. On the other hand, (write 'a' (state("abc", "xyz")) = nonnull(state("abc", "xyza")).

When a function produces a null value, it indicates that it has encountered an error, and cannot produce a meaningful value. A null value can only be produced when the result type is a **maybe** type.

Expressions that support monads have the following forms and meanings.

A ; *B*

> Same as (seq *A* *B*). See below for the definition of seq. The idea is to run function *A* in the initial state, and to feed the state that *A* produces into *B*. So this simulates sequencing in an imperative language.

return *A*

> Same as (\s -> nonnull(s,*A*)). The idea is to produce the value of *A* as the value of a computation.

(*x* <- *A*) ; *B*

> Same as (bindseq (*A*) (\x -> *B*)), where bindseq is defined below. The idea is to compute *A* in the initial

state, to take the value produced by *A* and name it *x*, and then to evaluate *B* in the state produced by *A*, with the given value of *x*.

The definitions of seq and bindseq are as follows.

```
type seq ::
      (state -> maybe state)
      -> (state -> maybe (state, x))
      -> (state -> maybe (state, x))
def seq x y s =
  let t = x s in
    case
      nullq t => null
    | else    => y(value t))
    end
  end
end

type bindseq ::
         (state -> maybe (state, x))
         -> (x -> (state
                         -> maybe (state, y)
         -> state
         -> maybe (state, y)

def bindseq x y s =
  let t = x s in
    case
      nullq t => null
    | else    => y (right(value t)) (left(value t))
    end
  end
end
```

Examples

Although the concatenation operator ++ is directly available, you could define your own list concatentation function as follows.

```
type cat:: list a -> list a > list a
def cat x y =
  case
    nilq x => y
  | else   => head x : cat (tail x) y
  end
end
```

Here is a function composition function. (compose f g) is a function which, when applied to x, produces f(g(x)).

```
type compose:: (b -> c) -> (a -> b) -> a -> c
def compose f g x = f(g x)
end
```

Here is an alternative definition of compose.

```
   type compose:: (b -> c) -> (a -> b) -> a -> c
   def compose f g = (\ x -> f(g x))
   end
```

Here is yet another definition of compose.

```
   type compose:: (b -> c) -> (a -> b) -> a -> c
   def compose = (\f -> (\g -> (\x -> f(g x))))
   end
```

Target machine

This will be supplied later.

Refinements

It is not necessary to produce an implementation of the entire language all at once. A short sequence of subsets will be provided. It is expected that not all students will complete all subsets.