Homework: Basic Interpreter

The high-level goal of this assignment is to implement an interpreter for a small, functional programming language. In the next assignment, you'll grow this interpreter with non-functional features.

1 Restrictions

You can use any features of OCaml you like. We strongly recommend avoiding objects. Imperative features aren't necessary, but can be helpful for implementing recursion.

2 Requirements

Write a program that takes one argument:

`./interp.d.byte` *program*

The argument *program* should be the name of a file that contains a program written using the grammar defined in fig. 6.1. Your interpreter may output the result of the program in any format you like. However, **if the result of an operation is not defined, the interpreter must exit with a non-zero exit code.** (In OCaml, if your program terminates with an unhandled exception or a false assertion, it wil exit with code 2.)

You must implement a call-by-value semantics, using the operational semantics from class as a guide (refer to the lecture notes). You must also implement *proper tail calls*.

The language that you need to implement is richer than we've discussed in class. A few things to note:

- The following big-step rule describes the semantics of the `fix` construct:

$$\textsc{Fix}\ \frac{e[x \mapsto \texttt{fix}\ x\texttt{->}\ e] \Downarrow v}{\texttt{fix}\ x\texttt{->}\ e \Downarrow v}$$

  The intuition behind this rule is that we evaluate the body e by substituting x with the original expression. Therefore, e can refer to itself via x.

- In a list value, the head and tail of the list must be evaluated to values themselves. For example, `(1 + 2) :: empty` is not a value. The same reasoning applies to records.

- You will be able to write several kinds of programs that don't make any sense. E.g., `true + false`, `if 3 then 4 else 5`, `x` (where `x` is a free identifier). In these situations, the interpreter should throw an exception and exit. Do not try to recover from these kinds of errors in any way.

3 Support Code

In the support code for the class, the `Interp_util` module defines an abstract syntax for the language and a parser for the concrete syntax shown in fig. 6.1. You are not strictly required to use the support code. In particular, you are free to copy it and modify it if you wish to extend the language in any way. However, any extensions you make must be compatible with the base language.

Expressions

e	::=	x	`Id x`	where x has letters, digits, and underscores
	|	n	`Const (Int n)`	where n is a decimal integer
	|	`true`	`Const (Bool true)`	Boolean true
	|	`false`	`Const (Bool false)`	Boolean false
	|	e_1 `+` e_2	`Op2 (Add,` e_1`,` e_2`)`	Integer addition, as defined in OCaml
	|	e_1 `-` e_2	`Op2 (Sub,` e_1`,` e_2`)`	Integer subtraction, as defined in OCaml
	|	e_1 `*` e_2	`Op2 (Mul,` e_1`,` e_2`)`	Integer multiplication, as defined in OCaml
	|	e_1 `/` e_2	`Op2 (Div,` e_1`,` e_2`)`	Integer division, as defined in OCaml
	|	e_1 `%` e_2	`Op2 (Mod,` e_1`,` e_2`)`	Modulus, as defined in OCaml
	|	e_1 `<` e_2	`Op2 (LT,` e_1`,` e_2`)`	Integer less-than, as defined in OCaml
	|	e_1 `>` e_2	`Op2 (GT,` e_1`,` e_2`)`	Intege greater-than, as defined in OCaml
	|	e_1 `==` e_2	`Op2 (Eq,` e_1`,` e_2`)`	Equality of booleans and integers
	|	`empty`	`Empty`	An empty list
	|	e_1 `::` e_2	`Cons (`e_1`,` e_2`)`	A list with e_1 as the head and e_2 as the tail
	|	`{`x_1`:` e_1`,` $\cdots$`,`x_n`:`e_n`}`	`Record [(`x_1`,`e_1`); ... (`x_n`,` e_n`)]`	A record with n named fields
	|	e`.`x	`GetField(`e`,`x`)`	The value of field x
	|	`head` e	`Head` e	Produces the head of the list e
	|	`tail` e	`Tail` e	Produces the tail of the list e
	|	`is_empty` e	`IsEmpty` e	Produces `true` if e is the empty list
	|	`(`e`)`	`e`	Parentheses
	|	e_1 e_2	`App (`e_1`,` e_1`)`	Function application
	|	`if` e_1 `then` e_2 `else` e_3	`If (`e_1`,` e_2`,` e_3`)`	Conditional (e_1 must be a boolean)
	|	`let` x `=` e_1 `in` e_2	`Let (`x`,` e_1`,` e_2`)`	Let binding
	|	`fun` x `->` e	`Fun (`x`,` e`)`	Function definition
	|	`fix` x `->` e	`Fix (`x`,` e`)`	Recursive function definition

Figure 6.1: The concrete syntax and abstract syntax of the language.