Homework: Type Checker

The goal of this assignment is to implement a type checker to complement the interpreter you've built so far.

1 Restrictions

None.

2 Requirements

Write a program that takes one argument:

`./tc.d.byte` *program*

The argument *program* should be the name of a file that contains a program written using the grammar defined in fig. 11.1. If the program is type-correct, your type-checker should exit normally with exit code 0. If there is a type error, it should exit with a non-zero exit code. (Therefore, you can output anything you want to standard out and standard error.)

3 Support Code

In the support code for the class, the *Tc_util* module defines an abstract syntax for the language and a parser for the concrete syntax shown in fig. 11.1. 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.

Types

τ	::=	`bool`	`TBool`	Type of booleans
	|	`int`	`TInt`	Type of integers
	|	τ_1`->`τ_2	`TFun (`τ_1`,` τ_2`)`	Type of a function
	|	{x_1`:` τ_1`,` $\cdots$`,`x_n`:`τ_n}	`TRecord [(`x_1`,`τ_1`); ... (`x_n`,` τ_n`)]`	Type of a record
	|	τ `list`	`TList` τ	Type of a list
	|	τ `array`	`TArr` τ	Type of an array
	|	`forall` x `.` τ	`TForall (`x`,` τ`)`	Quantified type
	|	x	`TId` x	Type identifier

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<`τ`>`	`Empty` τ	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`:`τ`) ->` e	`Fun (`x`,` τ`,`e`)`	Function definition
	|	`fix (`x`:`τ`) ->` e	`Fix (`x`,` τ`,`e`)`	Recursive unction definition
	|	`array(`e_1`,`e_2`)`	`MkArray (`e_1`,` e_2`)`	Allocates an array of length e_1 with e_2 at every index
	|	e_1`[`e_2`]`	`GetArray (`e_1`,` e_2`)`	Produces the value at index e_2 in the array e_1
	|	e_1`[`e_2`]` `=` e_3	`SetArray (`e_1`,` e_2`,` e_3`)`	In the array e_1, set the value at index e_2 to the value of of e_3 and returns that value
	|	`tfun` x `.` e	`TypFun (`x`,` e`)`	Type function
	|	e`<`τ`>`	`TypApp (`e`,` τ`)`	Type application

Figure 11.1: The concrete syntax and abstract syntax of the language.