CS:3820 Programming Language Concepts
Spring 2019

Course Project

Due: Friday, May 10 by 11:59pm

The grade for this team project will be given on an individual basis. All students in a team must also submit an evaluation on how well they and their teammate performed as team members. Each evaluation is confidential and will be incorporated in the calculation of the grade.

Be sure to review the syllabus for details about this course's cheating policy. In particular, be mindful that the whole team is responsible for the submission, regardless of how the work is divided among the team members.

Expand the accompanying archive `project.zip` and put the extracted folder `project` on your desktop. The folder contains a few files that you will need. Write your solutions as instructed below. Then compress `project` to a zip archive called `projectsol.zip` and submit the archive. *Make sure you submit the new zip file with your solution, not the original one!*

The submission policy for the code and the evaluations is the same as with team homework assignments. In particular, *only one person per team should submit the code.*

Note: Your files *must compile* with no syntax/type errors. You may receive serious penalties for code that does not compile correctly.

1 The PLC Language

In this project you will develop in F# an interpreter and related code for an extension of a variation of the PLC language introduced in Hw6. We will also refer to this new language as the PLC language. The language incorporates several of the programming concepts studied during the course. It is purely functional, strict, statically-typed, lexically-scoped, and higher-order. With respect to the version from Hw6, this version of PLC has more features, including a sequence type and related primitive functions. The sequence type of PLC is very similar to the list type of F#, with values consisting of an ordered, immutable series of elements of the same type. Other features include anonymous functions, simple pattern matching, and a print command. Also, the concrete and abstract syntax for PLC types and expressions is slightly different, but mostly very similar to that of micro-ML.

Main limitations with respect to real world functional languages, introduced for simplicity, are that there are no commands to read input from the console or files; functions are monomorphic and can be recursive but not mutually recursive; formal parameters of functions must be explicitly typed; recursive functions must declare their return type; and pattern matching is restricted to

```
fun inc (Int x) = x + 1;
fun add (Int x, Int y) = x + y;
fun cadd (Int x) = fn (Int y) => x + y end;
var y = add(3, inc(4));
var x = cadd(3)(7-y);
var z = x * 3;
fun rec fac (Int n) : Int =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> n * fac(n - 1)
  end
;
print x; print y;
x :: y :: z :: fac(z) :: ([Int] [])
```

Figure 1: A PLC program.

value comparison, i.e. it amounts to syntax sugar for a series of if-then-else statements. Finally, important basic types such as characters and strings or structured types such as algebraic datatypes and records are missing.

Figure 1 contains an example of a PLC program. The program defines a non-recursive first-order function `inc` of type `Int -> Int`; a non-recursive first-order function `add` of type `(Int,Int) -> Int`; a non-recursive higher-order function `highAdd`; variables `x`, `y` and `z`; and a recursive first-order function `fact`. The scope of each of these functions and variables includes the declarations and expressions that follow. In the example, the expressions after the declaration of `fact` are also separated by semicolons. When used with expressions, semicolon is, as in F#, a right-associative binary operator such that e_1 `;` e_2 evaluates to the value of e_2 for all expressions e_1 and e_2. In the example program, the first expression prints to the console the value of `x` and `y` and then returns the list consisting of the values of `x`, `y`, `z` , `fact(z)` and `y`. Function `highAdd` takes an integer `x` and returns the anonymous function `fn (Int y) => x + y end`, which in turn takes an integer `y` and returns the value of `x + y`. Function `fact` is the usual factorial function whose input and output values are explicitly declared to be of type `Int`.

In this version of PLC, function declarations must include the output type *only* if the function is recursive. Declarations of such functions need the `rec` qualifier after the `fun` keyword. Since PLC has anonymous functions, declarations of non-recursive functions are in fact syntactic sugar. That is, a program of the form

$$\texttt{fun } f(t\ x) \texttt{ = } e \texttt{ ; } e_1$$

is treated as the program

$$\texttt{var } f \texttt{ = fn (} t\ x \texttt{) => } e \texttt{ end ; } e_1$$

where f becomes a variable of higher-order type t `->` t_e (with t_e being the type of e) whose value is the anonymous function `fn (`t x`) =>` e `end`. The only true function declarations are those of recursive functions then.

```
var E = ([Int] []);
fun reverse ([Int] s) = {
  fun rec rev ([Int] s1, [Int] s2): [Int] =
    match s1 with
    | E -> s2
    | _ -> {
             var h = hd(s1);
             var t = tl(s1);
             rev(t, h::s2)
           }
    end
  ;
  rev(s, E)
};
reverse (1::2::3::E)
```

Figure 2: A PLC program with locally defined functions.

One restriction on the use of `;` is that (variable or function) declarations cannot follow expressions unless they are included in a brace-delimited block. For example;

```
1 - 3; var x = 4; 2 * x
```

is not allowed whereas

```
1 - 3; {var x = 4; 2 * x}
```

is. In any case, the last argument of `;` must be an expression, it cannot be a declaration.

Sequences of declarations and expressions enclosed in braces are treated as atomic expressions, which means that they can go anywhere an expression can go. This allows one for instance to declare local variables and functions within another function, as in the program in Figure 3.

2 Types, type annotations and static typing

Sequences in PLC are essentially the same as lists in F#, with `[]` denoting the empty sequence constant and `::` denoting the sequence constructor. Note, however, that the empty sequence must be explicitly typed anywhere it occurs, as shown in the programs of Figures 1– 3. The reason is that this makes type checking considerably easier to implement. It is the same reason formal parameters of functions must be explicitly typed and recursive functions must declare their return type. The latter simplifies the type checking of the function's body, which includes occurrences of the function name (in recursive calls).

Since the language is higher-order, we can define and use in it the usual combinators, with the only restriction that they cannot be polymorphic.[1] Examples of such functions are provided in Figure 2. The function `map` is the usual one except that it is restricted to integers sequences as input and as output, and has a more verbose declaration than in F#.

[1] This is truly limiting, because now one needs for instance to define a `map` function for each possible concrete instance, such as `(Int -> Int) -> [Int] -> [Int]`, of the parametric type `('a -> 'b) -> ['a] -> ['a]` that `map` could have if polymorphism was allowed as in F#. This restriction too is to simplify type checking.

```
fun twice (Int -> Int f) = fn (Int x) => f(f(x)) end ;
fun rec map (Int -> Int f) : ([Int] -> [Int]) =
  fn ([Int] s) =>
    if ise(s) then s else f(hd(s)) :: map(f)(tl(s))
  end ;
fun square (Int x) = x * x ;
fun inc (Int x) = x + 1 ;
var E = ([Int] []) ;
var s1 = map (fn (Int x) => 2*x end) (10::20::30::E) ;
var s2 = map (twice(inc)) (s1) ;
(s1, s2)
```

Figure 3: A PLC program with higher-order combinators.

2.1 Types and operators

The language has the following types and operations on them. Your interpreter should support all of them.

Nil type The type `Nil`, similar to **`unit`** in F#, contains a single value. Predefined operators dealing with `Nil` values are: `() : Nil`, the only value of this type, and **`print`** `:` t `-> Nil`, for any type t. The latter function always returns `()` but has the side effect of printing to the console (standard output) a textual representation of its input value.

Boolean type The type `Bool` is the usual Boolean type. In addition to the constants **`true`** and **`false`**, it has the predefined operators `&& : (Bool, Bool) -> Bool` for Boolean conjunction and `! : Bool -> Bool` for Boolean negation. Two more operators are `=` and `!=`, both of type `(`t`,` t`) -> Bool` for any *equality* type t (see below), respectively for equality and disequality comparison.

Integer type The type `Int` is the usual integer type whose constants are all the numerals. It has the usual infix binary operators `+`, `-`, `*`, `/`, `<`, and `<=` with the expected meaning. The first four have type `(Int,Int) -> Int`. The last two have type `(Int, Int) -> Bool`. The `-` operator is also unary, with type `Int -> Int`.

List types For any PLC types t_1, ..., t_n with $n > 1$ it is possible to construct lists of type `(`t_1`, ...,` t_n`)`. The list constructor is the multi-arity mixfix operator `(_, ..., _)`. For all $n > 0$, $i \in \{1, \ldots, n\}$ and types t_1, ..., t_n, there is also a postfix element selector `[`i`] : (`t_1`, ...,` t_n`) ->` t_i that returns the ith element of its input list.

Function types Functions that take an input of type t_1 and produce an output of type t_2 have type t_1 `->` t_2. The arrow operator `->` is right-associative.

Sequence types For any PLC type t it is possible to construct sequences of type `[`t`]`. Note that this means that it is possible to construct sequences of sequences, sequences of lists, and so on. The predefined, and polymorphic, operators dealing with sequence values are listed below.

- `[]` : `[`t`]`, for any type t. The empty sequence of elements of type t.

- `::` : `(t, [t]) -> [t]`, for any type t. The infix, right-associative sequence construction operator.
- `ise` : `[t] -> Bool`, for any type t. Returns `true` if the input sequence is empty and `false` otherwise.
- `hd` : `[t] -> t`, for any type t. Returns the head of the input sequence if the input is not empty, and raises an exception otherwise.
- `tl` : `[t] -> [t]`, for any type t. Returns the tail of the input sequence if the input is not empty, and raises an exception otherwise.

Equality types These are the types with no occurrences of `->` in them. They are defined inductively as follows: (*i*) `Bool`, `Int`, and `Nil` are equality types; (*ii*) if t is an equality type, so is `[t]`; (*iii*) if $t_1, \ldots, t_n$ with $n > 1$ are equality types, so is `(t1, ..., tn)`; (*iv*) nothing else is an equality type. Recall that `=` and `!=` apply only to values of an equality type.

An additional predefined infix operator is `;` which has type `(t1, t2) -> t2` for any types t_1 and t_2. It works exactly as in F# by evaluating, in order, each of its arguments and returning the value of its second argument. In PLC, it is most useful when the first argument contains applications of the `print` function.

3 Concrete Syntax

The concrete syntax of PLC is described by the grammar rules below[2], where non-terminal symbols are written in angular brackets and the top symbol is `<prog>`.

3.1 Production rules

```
<prog> ::= <expr> | <decl> ; <prog>

<decl> ::=
    var <name> = <expr>
  | fun <name> <args> = <expr>
  | fun rec <name> <args> : <type> = <expr>

<expr> ::=
    <atomic expr>                                 atomic expression
  | <app expr>                                    function application
  | if <expr> then <expr> else <expr>             conditional expression
  | match <expr> with <matchexpr>                 match expression
  | ! <expr>                                      unary operator application
  | - <expr>
  | hd <expr>
  | tl <expr>
  | ise <expr>
  | print <expr>
  | <expr> + <expr>                               binary operator application
```

[2]The production rules for ¡args¿ are not complete because this is part of Homework 6. The complete concrete syntax will be posted after Homework 6's deadline.

```
  | <expr> - <expr>
  | <expr> * <expr>
  | <expr> / <expr>
  | <expr> = <expr>
  | <expr> != <expr>
  | <expr> < <expr>
  | <expr> <= <expr>
  | <expr> :: <expr>
  | <expr> ; <expr>
  | <expr> [ <nat> ]

<atomic expr> ::=
    <const>                                   constant literal
  | <name>                                    function, variable or parameter name
  | { <prog> }                                local scope block
  | ( <expr> )                                parenthesized expression
  | ( <comps> )                               list
  | fn <args> => <expr> end                   anonymous function

<app expr> ::=                                function application
    <atomic expr> <atomic expr>
  | <app expr> <atomic expr>

<const> ::=
    true | false
  | <nat>                                     numerals
  | ( )                                       nil value
  | ( <type> [ ] )                            type-annotated empty sequence

<comps> ::=                                   list components
    <expr> , <expr>
  | <expr> , <comps>

<matchexpr> ::=                               match cases
    end
  | '|' <condexpr> -> <expr> <matchexpr>

<condexpr> ::=                                values to be matched against
    <expr>
  | '_'

<args> ::=                                    function arguments
    ( )
  | ( <params> )

<params> ::=
    <typed var>
  | <typed var> , <params>

<typed var> ::= <type> <name>                 typed variable
```

```
<type> ::=
     <atomic type>
   | ( <types> )                                     list type
   | [ <type> ]                                      sequence type
   | <type> -> <type>                                function type

<atomic type> ::=
     Nil                                             Nil type
   | Bool                                            Boolean type
   | Int                                             integer type
   | ( <type> )

<types> ::=
     <type> , <type>
   | <type> , <types>
```

3.2 Lexical rules

The non-terminal `<name>` is a token defined by the regular expression

```
['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '_' '0'-'9']*
```

excluding the following names, which are keywords:

```
Bool else end false fn fun hd if Int ise
match Nil print rec then tl true var with _
```

The non-terminal `<nat>` is a token defined by the regular expression `[0-9]+`.

3.3 Operator precedence

The various operators and keywords have the following precedence, from lower to higher, with operators on the same line having the same precedence.

`; ->`	(right-associative)
`if`	(non-associative)
`else`	(left-associative)
`&&`	(left-associative)
`= !=`	(left-associative)
`< <=`	(left-associative)
`::`	(right-associative)
`+ -`	(left-associative)
`* /`	(left-associative)
`not hd tl ise print` f	(non-associative)
`[`	(left-associative)

where f is any user-defined function name.

```
type plcType =
    | IntT                              //  Int
    | BoolT                             //  Bool
    | FunT  of plcType * plcType        //  type -> type
    | ListT of plcType list             //  Nil and (type, ..., type)
    | SeqT  of plcType                  // [type]

type expr =
    | ConI   of int                                //  integer constants
    | ConB   of bool                               //  Boolean constants
    | ESeq   of plcType                            //  typed empty sequence constant
    | Var    of string                             //  variables
    | Let    of string * expr * expr               //  expressions with variable declaration
    | Letrec of string * plcType * string          //  expressions with recursive function decl.
                 * plcType * expr * expr
    | Prim1  of string * expr                      //  unary operators
    | Prim2  of string * expr * expr               //  binary operators
    | If     of expr * expr * expr                 //  if construct
    | Match  of expr * (expr option * expr) list   //  match construct
    | Call   of expr * expr                        //  function application
    | List   of expr list                          //  Nil Constant / list construction
    | Item   of int * expr                         //  List selector application
    | Anon   of plcType * string * expr            //  anonymous function

type plcVal =
    | BoolV of bool                                //  Booleans
    | IntV  of int                                 //  integers
    | ListV of plcVal list                         //  lists
    | SeqV  of plcVal list                         //  sequences
    | Clos  of string * string * expr * plcVal env //  closures
```

Figure 4: Abstract syntax for PLC programs.

4 Abstract Syntax

For uniformity, and to make your task easier, we fix an abstract syntax for PLC types, expressions and values as the F# algebraic data types in Figure 4. You must use this abstract syntax in your implementation. The abstract syntax tree is also available in module `Absyn`.

4.1 Types

F# terms of type `plcType` are used to encode PLC types. Here are examples of PLC code and their corresponding abstract syntax:

Concrete syntax	Abstract syntax
`Int`	`IntT`
`Nil`	`ListT []`
`Int -> Int`	`FunT (IntT, IntT)`
`Int -> Int -> Bool`	`FunT (IntT, FunT (IntT, BooT))`
`(Int -> Int) -> Bool`	`FunT (FunT (IntT, IntT), BooT)`
`(Int, Int, Bool)`	`ListT [IntT; IntT; BooT]`
`(Int, Int) -> Bool`	`FunT (ListT [IntT; IntT], BooT)`
`[Int]`	`SeqT IntT`
`[(Bool,Int)]`	`SeqT (List [BooT; IntT])`

Note that the `plcType` constructor `ListT` is used to represent both the `Nil` type, with `ListT []`, and list types, with `ListT [`t_1`; ...; `t_n`]` for $n > 1$.

4.2 Expressions

F# terms of type `exp` are used to encode PLC programs and expressions. Here are examples of PLC code and their corresponding abstract syntax:

<table>
<tr><th>Concrete syntax</th><th>Abstract syntax</th></tr>
<tr><td><code>15</code></td><td><code>ConI 15</code></td></tr>
<tr><td><code>true</code></td><td><code>ConB true</code></td></tr>
<tr><td><code>()</code></td><td><code>List []</code></td></tr>
<tr><td><code>(6, false)</code></td><td><code>List [ConI 6; ConB false]</code></td></tr>
<tr><td><code>(6, false)[1]</code></td><td><code>Item (1, List [ConI 6; ConB false])</code></td></tr>
<tr><td><code>([Bool] [])</code></td><td><code>ESeq (SeqT BoolT)</code></td></tr>
<tr><td><code>print x; true</code></td><td><code>Prim2 (";", Prim1 ("print", Var "x"), ConB true)</code></td></tr>
<tr><td><code>3::7::t</code></td><td><code>Prim2 ("::", ConI 3, Prim2 ("::", ConI 7, Var "t"))</code></td></tr>
<tr><td><code>fn (Int x) => -x end</code></td><td><code>Anon (IntT, "x", Prim1("-", Var "x"))</code></td></tr>
<tr><td><code>var x = 9; x + 1</code></td><td><code>Let ("x", ConI 9, Prim2 ("+", Var "x", ConI 1))</code></td></tr>
<tr><td><code>fun f(Int x) = x; f(1)</code></td><td><code>Let ("f", Anon (IntT, "x", Var "x"), Call ("f", ConI 1))</code></td></tr>
<tr><td><pre>match x with
| 0 -> 1
| _ -> -1
end</pre></td><td><pre>Match (Var "x",
        [(Some (ConI 0), ConI 1);
           (None, Prim1 ("-",ConI 1))])</pre></td></tr>
<tr><td><pre>fun rec f(Int n) =
  if n <= 0 then 0
  else n + f(n-1) ;
f(5)</pre></td><td><pre>Letrec ("f", IntT, "n",
  If (Prim2 ("<=", Var "n", ConI 0), ConI 0,
    IntT, Prim2 ("+", Var "n", Call (Var "f", ...))),
    Call (Var "f", ConI 5))</pre></td></tr>
</table>

The `List` constructor, which takes a list of expressions as arguments is used to represent list expressions. It is also used to represent the `Nil` expression `()`, as `List []`. Note that the empty sequence constant `ESeq` carries the sequence type with it, which is needed for type checking. Also note that `[i]`, represented by the `Item` constructor, is treated as binary operator for convenience; however, its second argument, `i`, must be a numeral.

Anonymous functions of the form `fn (`t x`) =>` e `end` are represented as `Anon (`t'`, `x`, `e'`)` where t' is the abstract syntax representation of type t and e' is the abstract representation of the function's body e. Otherwise, the conversion to abstract syntax should be generally done as in

Hw6. In particular, multi-argument functions should also be converted as in Hw6, using nested `Let` expressions.

4.3 Values

F# terms of type `plcValue` are used to encode PLC values. The PLC interpreter is essentially a converter from `expr` terms to `plcValue` terms. Here are examples of such conversions.

	Expression
1.	`ConI 15`
2.	`ConB true`
3.	`List []`
4.	`List [ConI 6; ConB false]`
5.	`Item (1, List [ConI 6; ConB false])`
6.	`ESeq (SeqT BoolT)`
7.	`Prim2 (";", Prim1 ("print", ConI 27), ConB true)`
8.	`Prim1 ("print", ConI 27)`
9.	`Prim2 ("::", ConI 3, Prim2 ("::", ConI 4, Prim2 ("::", ConI 5, ESeq (SeqT IntT))))`
10.	`Anon (IntT, "x", Prim1("-", Var "x"))`
11.	`Let ("x", ConI 9, Prim2 ("+", Var "x", ConI 1))`
12.	`Let ("f", Anon (Int, "x", Var "x"), Call ("f", ConI 1))`

	Value
1.	`IntV 15`
2.	`BoolV true`
3.	`ListV []`
4.	`ListV [IntV 6; BoolV false]`
5.	`IntV 6`
6.	`SeqV []`
7.	`BoolV true`
8.	`ListV []`
9.	`SeqV [3; 4; 5]`
10.	`Clos ("", "x", Prim1("-", Var "x")), [])` (in case of an empty environment)
11.	`IntV 10`
12.	`IntV 1`

Anonymous function expressions of the form `Anon (`*t*`, `*x*`, `*e*`)` should evaluate to the value `Clos ("", `*x*`, `*e*`, `*env*`)` where *env* is the current environment.

With expressions of the form `Prim1("print", `*e*`)`, the interpreter should first evaluate *e* to some value *v*, convert *v* to a string representation in concrete syntax, and then print that string to the standard output followed by a new line character. For the string conversion, you can use the helper function `val2string : plcVal -> string` already provided in module `Absyn`.

What other well-typed PLC expressions should evaluate to should be clear from Hw6. If you are not clear about specific cases, please ask the instructors.

5 Implementation

Your implementation of PLC should be divided in the following F# modules. Each module should be in its own file, with the same name and with extension `.fs`. You are required to follow this modularization both for your own sake, and to ease our evaluation of your code.

- `Environ`
  This module defines a generic environment type and associated `lookup` function. It is already provided in the file `project/Environ.fs` in `project.zip`. You will need instances of that type and will use `lookup` in the type checker and in the interpreter.

- `Absyn`
  This module defines the abstract syntax. It is already provided in the file `project/Absyn.fs` in `project.zip`. It contains the helper function `val2string` that can be use to implement `print`.

- `PlcParserAux`
  This module defines a few helper functions for the parser. As in Hw6, its implementation in the file `Project/PlcParserAux.fs` is incomplete and must be completed by you.

- `PlcParser`
  This module contains the parser for the PLC language. You should generate it with FSYacc in a file called `PlcParser.fs` from the provided file `PlcParser.fsy` which contains a partial FSYacc specification of the language. You are to complete the specification in `PlcParser.fsy` by adding production rules. *Do not change any of the already defined tokens and their precedence.*

- `Lexer`
  This module contains the lexer for the PLC language. You should generate it with FSLex from a file name `PlcLexer.fsl` that you have to write. That file should use the tokens defined in `PlcParser.fsy` and recognize the operators and keywords of PLC. Your lexer may support comments, which in PLC have the form `(* ... *)`, but it does not need to.

- `Parse`
  This module defines a function `fromString`, to parse a PLC program from a string, and `fromFile`, to parse a PLC programs from a text file. You can use these functions to test your parser. The module is already provided for you in file `Parse.fs`.

- `PlcChecker`
  This module contains the type checker. It should provide a function `teval : expr -> plcType env -> plcType` that, given an abstract syntax expression e and a type environment for the free variables in e, if any, returns the type of e if e is well-typed and fails (with `failwith`) otherwise. You are to implement `teval` in file `PlcChecker.fs` following the typing rules specified in Appendix A.

- `PlcInterp`
  This module contains the interpreter. It should provide a function `eval : expr -> plcValue env -> plcValue` that, given a *well-typed* expression e and a value environment for the free variables of e, if any, returns the value of e in that environment. You are to implement `eval` in file `PlcInterp.fs`.

  Note that it is expected that `eval` will diverge (never returning a value) if e denotes a non-terminating computation; for instance, if e comes from a program like
  `fun rec f(Int x):Int = f(x - 1); f(0)`.

- `Plc`
  This module defines a function `run : expr -> string` that takes an abstract syntax expression e, type checks the expression with `teval`, evaluates it with `eval`, and then returns a string containing the value and type of e in concrete syntax. It is already provided for you in file `Plc.fs`. You can use it together with `parse.fromString` or `parse.fromFile` to test your implementation of the type checker and the interpreter.

For your convenience, the archive `project.zip` contains also a `bin` folder with the FSLex and FSYacc executables.

A Typing rules for PLC

In the following, x denotes variable/function names; n denotes numerals; e, e_1, e_2 denote PLC expressions; s, t, t_i denote PLC types; ρ denotes a type environment, that is, a partial mapping from variable/function names to types; $\rho[x \mapsto t]$ denotes the environment that maps x to t and is otherwise identical to ρ; $\mathit{type}(e,\ \rho) = t$ abbreviates the statement: "the type of expression e in environment ρ is t."

The rules below define the type system of PLC. An expression e is well typed and has type t in a typing environment ρ if and only if you can conclude $\mathit{type}(e,\ \rho) = t$ according to these rules.

1. $\mathit{type}(x,\ \rho) = \rho(x)$
2. $\mathit{type}(n,\ \rho) = \texttt{Int}$
3. $\mathit{type}(\texttt{true},\ \rho) = \texttt{Bool}$
4. $\mathit{type}(\texttt{false},\ \rho) = \texttt{Bool}$
5. $\mathit{type}(\texttt{()},\ \rho) = \texttt{Nil}$
6. $\mathit{type}(\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)},\ \rho) = \texttt{(}t_1\texttt{,}\ \ldots\texttt{,}\ t_n\texttt{)}$ if $n > 1$ and $\mathit{type}(e_i,\ \rho) = t_i$ for all $i = 1, \ldots, n$
7. $\mathit{type}(\texttt{(}t\ \texttt{[])},\ \rho) = t$ if t is a sequence type.
8. $\mathit{type}(\texttt{var}\ x\ \texttt{=}\ e_1\ \texttt{;}\ e_2,\ \rho) = t_2$ if
   $\mathit{type}(e_1,\ \rho) = t_1$ and $\mathit{type}(e_2,\ \rho[\texttt{x} \mapsto t_1]) = t_2$ for some type t_1
9. $\mathit{type}(\texttt{fun rec}\ f\ \texttt{(}t\ x\texttt{)}\ \texttt{:}\ t_1\ \texttt{=}\ e_1\ \texttt{;}\ e_2,\ \rho) = t_2$
   if $\mathit{type}(e_1,\ \rho[f \mapsto t\ \texttt{->}\ t_1][x \mapsto t]) = t_1$ and $\mathit{type}(e_2,\ \rho[f \mapsto t\ \texttt{->}\ t_1]) = t_2$
10. $\mathit{type}(\texttt{fn}\ \texttt{(}s\ x\texttt{)}\ \texttt{=>}\ e\ \texttt{end},\ \rho) = s\ \texttt{->}\ t$ if $\mathit{type}(e,\ \rho[x \mapsto s]) = t$
11. $\mathit{type}(e_2\texttt{(}e_1\texttt{)},\ \rho) = t_2$ if $\mathit{type}(e_2,\ \rho) = t_1\ \texttt{->}\ t_2$ and $\mathit{type}(e_1,\ \rho) = t_1$ for some type t_1
12. $\mathit{type}(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\ \rho) = t$ if $\mathit{type}(e,\ \rho) = \texttt{Bool}$ and $\mathit{type}(e_1,\ \rho) = \mathit{type}(e_2,\ \rho) = t$
13. $\mathit{type}(\texttt{match}\ e\ \texttt{with}\ \texttt{|}\ e_1\ \texttt{->}\ r_1\ \texttt{|}\ \ldots\texttt{|}\ e_n\ \texttt{->}\ r_n,\ \rho) = t$ if
    (a) $\mathit{type}(e,\ \rho) = \mathit{type}(e_i,\ \rho)$, for each e_i different from '_', and
    (b) $\mathit{type}(r_1,\ \rho) = \ldots = \mathit{type}(r_n,\ \rho) = t$

14. $\textit{type}(\texttt{!}e,\ \rho) = \texttt{Bool}$ if $\textit{type}(e,\ \rho) = \texttt{Bool}$

15. $\textit{type}(\texttt{-}e,\ \rho) = \texttt{Int}$ if $\textit{type}(e,\ \rho) = \texttt{Int}$

16. $\textit{type}(\texttt{hd(}e\texttt{)},\ \rho) = t$ if $\textit{type}(e,\ \rho) = \texttt{[}t\texttt{]}$

17. $\textit{type}(\texttt{tl(}e\texttt{)},\ \rho) = \texttt{[}t\texttt{]}$ if $\textit{type}(e,\ \rho) = \texttt{[}t\texttt{]}$

18. $\textit{type}(\texttt{ise(}e\texttt{)},\ \rho) = \texttt{Bool}$ if $\textit{type}(e,\ \rho) = \texttt{[}t\texttt{]}$ for some type t

19. $\textit{type}(\texttt{print(}e\texttt{)},\ \rho) = \texttt{Nil}$ if $\textit{type}(e,\ \rho) = t$ for some type t

20. $\textit{type}(e_1\ \texttt{\&\&}\ e_2,\ \rho) = \texttt{Bool}$ if $\textit{type}(e_1,\ \rho) = \textit{type}(e_2,\ \rho) = \texttt{Bool}$

21. $\textit{type}(e_1\ \texttt{::}\ e_2,\ \rho) = \texttt{[}t\texttt{]}$ if $\textit{type}(e_1,\ \rho) = t$ and $\textit{type}(e_2,\ \rho) = \texttt{[}t\texttt{]}$

22. $\textit{type}(e_1\ \textit{op}\ e_2,\ \rho) = \texttt{Int}$ if $\textit{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$ and $\textit{type}(e_1,\ \rho) = \textit{type}(e_2,\ \rho) = \texttt{Int}$

23. $\textit{type}(e_1\ \textit{op}\ e_2,\ \rho) = \texttt{Bool}$ if $\textit{op} \in \{\texttt{<}, \texttt{<=}\}$ and $\textit{type}(e_1,\ \rho) = \textit{type}(e_2,\ \rho) = \texttt{Int}$

24. $\textit{type}(e_1\ \textit{op}\ e_2,\ \rho) = \texttt{Bool}$ if $\textit{op} \in \{\texttt{=}, \texttt{!=}\}$ and $\textit{type}(e_1,\ \rho) = \textit{type}(e_2,\ \rho) = t$ for some **equality type** t

25. $\textit{type}(e\ \texttt{[}i\texttt{]},\ \rho) = t_i$ if $\textit{type}(e,\ \rho) = \texttt{(}t_1\texttt{,}\ \ldots\texttt{,}\ t_n\texttt{)}$ for some $n > 1$ and types $t_1, \ldots, t_n$, and $i \in \{1, \ldots, n\}$

26. $\textit{type}(e_1\ \texttt{;}\ e_2,\ \rho) = t_2$ if $\textit{type}(e_1,\ \rho) = t_1$ for some type t and $\textit{type}(e_2,\ \rho) = t_2$