CS:3820 Programming Language Concepts
Fall 2016

Course Project

Due: Friday, Dec 16 by 11:30pm

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.

Expand the accompanying archive `Project.zip` and put the extracted folder `Project` on the Windows (or MacOs) 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 `Project.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 does not compile correctly.

1 The HawkFun Language

In this project you will develop an interpreter **in F#** for a toy language called HawkFun that incorporates several of the programming concepts studied during the course. The language is purely functional, strict, statically-typed, lexically-scoped, and higher-order. Operationally, it is very similar to the language HighFun seen in class but with a somewhat different syntax and more features, including a list and a unit type and related primitive functions, anonymous functions, and a print command.

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 only unary and monomorphic, functions can be recursive but not mutually recursive, several important basic types (such as strings) or type structures (such as algebraic datatypes and records) are missing, all formal parameters of functions must be explicitly typed.

Figure 1 contains an example of a HawkFun program. The program defines a non-recursive first-order function `inc`, a non-recursive higher-order function `add`, variables `y` and `z`, and a recursive first-order function `fac`. The scope of each of these functions and variables includes the declarations that follow and the expression between `in` and `end`. In the example, the latter prints to the console the value of `x`, `y` and `fac 5`, and then returns the list consisting of the values of `x` and `y`. Function

```
local
  fun add (x : int) = fn (y : int) => x + y end
  fun inc (x : int) = x + 1
  var y = add 3 (inc 4)
  var z = y * 3
  fun rec fac (n : int) : int =
    if n = 0 then 1 else n * (fac (n - 1))
in
  print x; print y; fac 5;
  x :: y :: ([] : int list)
end
```

Figure 1: A HawkFun program.

`add` takes an integer `x` and returns the anonymous function `fn (y:int) => x + y end`, which in turn takes an integer `y` and returns the result of `x + y`. Function `fac` is the usual factorial function who input and output are explicitly declared to be of type `int`. Note how, contrary to HighFun, it is possible to define several local functions and variables in sequence. This is, however, just syntactic sugar for nested local declarations. For instance, a program of the form

```
local
  y = e1
  fun f (x : t) = e2
  z = e3
in
  e4
end
```

has the same meaning (and is converted to the same abstract syntax) as the program

```
local
  y = e1
in
  local
    fun f (x : t) = e2
  in
    local
      z = e3
    in
      e4
    end
  end
end
```

In fact, since we have anonymous functions, declarations of non-recursive function declaration are syntactic sugar too. For instance, a program of the form

```
local
  var el = ([] : int list)
  fun reverse (l : int list) =
    local
      fun rec rev (l1 : int list) : int list =
        fn (l2 : int list) =>
          if l1 = el then l2 else rev (tl l1) ((hd l1)::l2)
        end
    in
      rev l el
    end
in
  reverse (1::2::3::el)
end
```

Figure 2: A HawkFun program with locally defined functions.

```
local
  fun f (x : t) = e
in
  e1
end
```

has the same meaning as the program

```
local
  var f = fn (x : t) => e end
in
  e1
end
```

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

The construct `local ... in ... end` constructs an expression and so 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 2.

2 Type annotations and typing

Note that the empty list `[]` occurring in the program above and in Figure 1 is explicitly typed to be an `int list`. Giving an explicit type to each occurrence of `[]` is required. 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 too, although it makes it impossible to define polymorphic functions, that is, functions with parametric types such as `'a list -> 'b list`. In recursive function declarations, even the output type of the function must be explicitly declared. This is to

```
local
  var twice = fn (f:int -> int) => fn (x:int) => f (f x) end end
  var compose = fn (f:int -> int) => fn (g:int -> int) => fn (x:int) =>
                  f (g x)
                end end end
  fun rec map (f: int -> int) : (int list -> int list) =
    fn (l: int list) =>
      if ise l then l else (f (hd l)) :: (map f (tl l))
    end

  fun square (x:int) = x * x
  fun inc (x:int) = x + 1
  var inc2 = twice inc

  var e = ([]:int list)

  var x = compose inc square 3
  var l = map (fn (x:int) => 2*x end) (1::2::3::e)
in
  x::l
end
```

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

simplify 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] An example of such functions is provided in Figure 3. The function `map` is the usual one except that it is restricted to integers lists as input and as output, and it has a much more verbose declaration. Its body uses the predefined list operators `ise`, `hd` and `tl`. The first returns true if the input is the empty list and false otherwise. The other two are the usual head and tail functions for lists.

2.1 Types and operators

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

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.

List types For any type t in the language it is possible to construct lists of type t `list`. Note that this means that it is possible to construct lists of lists, among others. Predefined (overloaded) operators dealing with list values are listed below, with their type.

[1] This is truly limiting, because now one needs for instance to define a `map` function for each possible concrete instantiation, such as `(int -> int) -> int list -> int list`, of the parametric type `('a -> 'b) -> 'a list -> 'b list` that `map` has in F#. But again, it simplifies type checking.

- `[]` : t `list`, for any type t. The empty list of elements of type t.
- `::` : t `->` t `list ->` t `list`, for any type t. The usual infix list construction operator.
- `ise` : t `list -> bool`, for any type t. Returns `true` if the input list is empty and `false` otherwise.
- `hd` : t `list ->` t, for any type t. Returns the head of the input list if the input is not empty, and raises an exception otherwise.
- `tl` : t `list ->` t `list`, for any type t. Returns the tail of the input list if the input is not empty, and raises an exception otherwise.

Unit type The type `unit`, as in F#, contains a single value. Predefined operators dealing with unit values are listed below, with their type.

- `null` : `unit`. This is the single value of this type.
- `print` : t `-> unit`, for any type t. This function always returns `null`. It has the side effect though 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 a predefined operator `not : bool -> bool` for Boolean negation. Two more operators are `=` and `<>`, both of type t `->` t `-> bool` for any *equality* type t (see below), the first for equality comparisons and the second for disequality.

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. Note that, like user-defined functions, these operators are technically unary. The first four have higher-order type `int -> int -> int`. The last two have higher-order type `int -> int -> bool`.

Equality types These are the types with no occurrences of `->` in them. They are defined inductively as follows: (*i*) `bool`, `int`, and `unit` are equality types; (*ii*) if t is an equality type, so is t `list`; (*iii*) nothing else is an equality type.

An additional predefined infix operator is `;` which has type t_1 `->` t_2 `->` t_2 for any types t_1 and t_2. It works exactly as in F# by returning the value of its second input. It is most useful when its first argument is an expression of the form `print` e.

3 Concrete Syntax

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

3.1 Production rules

```
<expr> ::=
    <atom expr>                                   atomic expression
  | <app expr>                                    function application
```

```
  | if <expr> then <expr> else <expr>             conditional expression
  | not <expr>                                    unary operator application
  | hd <expr>
  | tl <expr>
  | ise <expr>
  | print <expr>
  | <expr> + <expr>                               binary operator application
  | <expr> - <expr>
  | <expr> * <expr>
  | <expr> / <expr>
  | <expr> = <expr>
  | <expr> <> <expr>
  | <expr> < <expr>
  | <expr> <= <expr>
  | <expr> :: <expr>
  | <expr> ; <expr>

<atomic expr> ::=
    <const>                                       constant literal
  | <name>                                        function, variable or parameter name
  | local <bindings> in <expr> end                local bindings
  | fn <typed name> => <expr> end                 anonymous function
  | ( <expr> )                                    parenthesized expression
  | ( [] : <type> )                               type-annotated empty list

<const> ::= <nat> | true | false | null

<typed name> ::= ( <name> : <type> )              typed name

<bindings> ::= <binding> | <binding> <bindings>

<binding> ::=
    var <name> = <expr>
  | fun <name> <typed name> = <expr>
  | fun rec <name> <typed name> : <type> = <expr>

<app expr> ::=
    <atomic expr> <atomic expr>                   one-argument function application
  | <app expr> <atomic expr>                      multi-argument function application

<type> ::=
     unit                                         unit type
   | bool                                         Boolean type
   | int                                          integer type
   | <type> -> <type>                             function type
   | <type> list                                  list type
   | ( <type> )
```

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 in int ise list
local not null print rec then true unit var
```

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)
`< <=`	(non-associative)
`::`	(right-associative)
`+ -`	(left-associative)
`* /`	(left-associative)
`not hd tl ise list print` *f*	(non-associative)

By *f* we mean any user-defined function name.

4 Abstract Syntax

For uniformity, and to make your task easier, we fix an abstract syntax for HawkFun in terms of the F# algebraic data types in Figure 4. Your parser should convert HawkFun concrete syntax to terms of these types, to be processed by the HawkFun type checker and interpreter. The first thing to notice is that abstract syntax expressions, i.e., values of (F#) type `expr`, explicitly carry their HawkFun type in them: they are pairs whose first element is the expression proper and the second is its type. Here are examples of HawkFun code and its corresponding abstract syntax:

Concrete syntax	**Abstract syntax**
1. `15`	1. `(Con 15, IntT)`
2. `true`	2. `(Con 1, BoolT)`
3. `null`	3. `(Con 0, UnitT)`
4. `([]:bool list)`	4. `(EListC, ListT BoolT)`
5. `x` with `x` an `int` var	5. `(Var "x", IntT)`
6. `fn (x:int) => x end`	6. `(Lam (("x", IntT), (Var "x", IntT)), ArrowT (IntT, IntT))`

```
type expr = expr1 * htype                    // typed expressions
and expr1 =
  | Con of int                               // integer, Boolean and unit constants
  | EListC                                   // empty list constant
  | Var of string                            // variables
  | Op1 of string * expr                     // unary operators
  | Op2 of string * expr * expr              // binary operators
  | If of expr * expr * expr                 // if construct
  | Let of binding * expr                    // expression with local declarations
  | Lam of tname * expr                      // anonymous function
  | Call of expr * expr                      // function application
and binding =
  | V of string * expr                       // variable declaration
  | F of string * tname * htype * expr       // recursive function declaration
and htype =
    AnyT                                     // dummy type
  | IntT | UnitT | BoolT                     // basic types
  | ArrowT of htype * htype                  // function type
  | ListT of htype                           // list type
and tname = string * htype                   // typed parameter
```

Figure 4: Abstract syntax for HawkFun programs.

Constants of basic types are all represented by expressions of the form `Con` n where n is an integer. Integer constants n are to be converted to `(Con` n`, IntT)`; the constants `true` and `false` to `(Con 1, BoolT)` and `(Con 0, BoolT)`, respectively; and the constant `null` to `(Con 0, UnitT)`. The (typed) empty list `([]:`t`)` is to be converted to `(EListC,` t'`)` where t' is the abstract syntax representation of the concrete syntax type t.

The abstract syntax expressions constructed with `Var`, `Op2`, `If`, and `Call` encode concrete syntax expressions as in HighFun. `Op1` is used to encode applications of unary operators.

`Lam` is used to encode anonymous functions. Specifically, a concrete syntax expression of the form `fn (`x`:`t`) =>` e `end` is encoded as `(Lam (("`x`",` t'`),` e'`),` t'_e`)` where t' encodes the type t of the input parameter x, e' encodes the body e of the function, and t'_e encodes the type t_e of e.

The constructor `Let` plays the role of both `Let` and `Letfun` in HighFun. It encodes `local` constructs that define a single variable or function, and uses values of the auxiliary type `binding` to distinguish the two cases, with the second one used only for *recursive* functions. For example, `local var x = false in not x end` is encoded as

```
(Let (V ("x", (Con 0, BoolT)), (Op1 ("not",(Var "x", BoolT)), BoolT)), BoolT)
```

Similarly, `local fun f (x:int) = x in (f 1) end` is encoded as

```
(Let (V ("f", (Lam (("x", IntT), (Var "x", IntT)), ArrowT (IntT, IntT))),
      (Call ((Var "f", ArrowT (IntT, IntT)), (Con 1, IntT)), IntT)),
 IntT)
```

as it if was `local var f = fn (x:int) => x end in (f 1) end`. In contrast,

```
local fun rec f (x:int):bool = ... in f 2 end
```

is encoded as

```
(Let (F ("f", ("x", IntT), BoolT, ...),
      (Call ((Var "f", ArrowT (IntT, BoolT)), (Con 2, IntT)), BoolT)),
 BoolT)
```

4.1 Type checking abstract syntax

As you can see, attaching the type to each expression and subexpression is rather verbose, and so space-inefficient, especially because in most cases type information is not needed at run time. We do this again for simplicity of implementation. In HawkFun, type information is needed at runtime by the `print` operator, because the way a value is printed out depends in its type. For instance, `Con 0` is printed as `0`, `false` or `null` depending on whether its attached type is `IntT`, `BoolT` or `UniT`, respectively.

Now, computing the type of an expression while parsing it is quite involved. So, we have added to `htype` a dummy type `AnyT`. You should write your parser so that it attaches type `AnyT` to any expression whose type is not obvious at parse time, the obvious cases being all constants and formal parameters. Once the full program has been successfully parsed, the resulting abstract syntax expression e will be given to type checking function that computes and returns another expression e' that is identical to e except that all occurrence of `AnyT` have been replaced by the proper actual type.

For instance, when parsing `local var x = false in not x end`, your parser should return just

```
(Let (V ("x", (Con 0, BoolT)), (Op1 ("not", (Var "x", AnyT)), AnyT)), AnyT)
```

Your type checker, given that expression, should then return

```
(Let (V ("x", (Con 0, BoolT)), (Op1 ("not", (Var "x", BoolT)), BoolT)), BoolT)
```

In the process of constructing the new expression, the type checker might realize that it is actually ill-typed. In that case, it should raise an exception (with `failwith`). For example, your parser may accept the (ill-typed) program `local var x = false in 2 * x end` and convert it to

```
(Let (V ("x", (Con 0, BoolT)) ,(Op2 ("*", (Con 2, IntT), (Var "x", AnyT)), AnyT)),
 AnyT)
```

The type checker should fail on this expression—which tries to multiply `false` by `2`.

The type checking rules for HawkFun are similar to those described in Chapter 4 of the textbook. A detailed description of these rules is provided in the appendix.

5 Evaluating expressions

Your interpreter will evaluate well-typed abstract syntax expressions to values of this F# type:

```
type value =
  | Int of int
  | List of value list
  | Closure of string option * string * expr * value env
```

This is similar to the value type used in the various interpreted toy languages seen in class, with the addition of a constructor for list values.

Abstract syntax expressions of a HawkFun basic type (`int`, `bool` and `unit`) should evaluate to values of the form `Int` n. Abstract syntax expressions of HawkFun list type (t `list`, for some type t) should evaluate to a value of the form `List` l where l is an F# list containing the evaluated elements of the HawkFun list. For instance, the abstract syntax expression

```
(Op2 ("::",(Con 1, IntT),
     (Op2 ("::",(Op2 ("+",(Con 1, IntT),(Con 2, IntT)), IntT),
          (EListC, ListT IntT)), ListT IntT)), ListT IntT)
```

which is the result of parsing and type checking the expression

```
1 :: (1 + 2) :: ([]:int list)
```

should be evaluated by the interpreter to `List [Int 1; Int 3]`. Note that, in general, the list l in `List` l can consist of elements themselves of the form `List` l' since the language permits nested lists, as in the expression

```
(1::2::3::([]:int list)) :: (4::3::([]:int list)) :: ([]:int list list)
```

Names of recursive functions should evaluate, as in the HighFun interpreter, to a value of the form `Closure (Some` f`,` x`,` e`,` env`)` where f is name of the function, x is the name of the input parameter, e is the body of the function, and env is the environment for the free (non-local) variables in e. Anonymous functions should evaluate, similarly, to a value of the form `Closure (None,` x`,` e`,` env`)` where f, x e and env are as above.[2]

Overall, apart from `print` expressions, what any particular well-typed HawkFun expression is supposed to evaluate to should be intuitively clear. If you are not clear about specific cases, please ask the instructors. As for print, see the next section.

6 Implementation

Your implementation of HawkFun 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.

- `Absyn`
  This module defines the abstract syntax. It is already provided in the file `Project/Absyn.fs` in `Project.zip`. It contains a helper function `htype` that you may find useful in other modules
- `Parser`
  This module contains the parser for the HawkFun language. You should generate it with FSYacc in a file called `Parser.fs` from a file `Parser.fsy` containing the FSYacc specification of the language. You are to write `Parser.fsy` and submit it with all other files.

[2] Note the use of the F# `option` type for the first argument of `Closure` given that anonymous functions have indeed no name.

- `Lexer`
  This module contains the lexer for the HawkFun language. You should generate it with FSLex from a file called `Lexer.fsl` containing the FSLex specification of the language. You are to write `Lexer.fsl` and submit it with all other files.

- `Parse`
  This module defines a function `fromString`, to parse a HawkFun program from a string, and `fromFile`, to parse a HawkFun programs from a text file. It is already provided for you.

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

- `TypeCheck`
  This module contains the type checker. It should provide a function `check : Absyn.expr -> Absyn.expr` that, given an expression e possibly containing instances of `AnyT`, returns an expression e' identical to e excepts that all of those instances have been replaced by the actual type of the associated expression. `check` should fail (with `failwith`) if e is ill-typed.

- `Inter`
  This module contains the interpreter. It should provide a function `eval : Absyn.expr -> value Env.env -> value` that, given a well-typed expression e (that is, one returned by `TypeCheck.check`) and a value environment for the free variables of e, if any, returns the value e evaluates to in that environment.

  With expressions of the form `print` e, function `eval` should first evaluate e to some value v, convert v to a string representation based on e's type, 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 `toString : value -> Absyn.htype -> string` already provided in `Inter.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
  `local fun rec f (x:int):int = f (x - 1) in f 0 end` .

7 Extra credit (optional)

Modify your implementation to supports a multiple argument syntax for functions, allowing for instance the declarations in the following program.

```
local
  var add = fn (x:int) (y:int) => x + y end
  fun sub (x:int) (y:int) = x - y
  fun rec rev (l1 : int list) (l2 : int list) : int list =
            if ise l1 then l2 else rev (tl l1) ((hd l1) :: l2)
in
  ...
end
```

A Typing rules for HawkFun

In the following, x denotes variable/function names; n denotes integer constants; e, e_1, e_2 denote HawkFun expressions; s, t, t_1, t_2 denote HawkFun types; $b_1, \ldots, b_n$ denote name bindings; 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 rule below define the type system of HawkFun. 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. $\mathit{type}(\texttt{null},\ \rho) = \texttt{unit}$
6. $\mathit{type}(\texttt{([] :}\ t\texttt{)},\ \rho) = t$ if t has the form $s\ \texttt{list}$ for some type s
7. $\mathit{type}(\texttt{local}\ b_1 \cdots b_n\ \texttt{in}\ e_2\ \texttt{end},\ \rho) =$
   $\mathit{type}(\texttt{local}\ b_1\ \texttt{in local}\ b_2 \cdots b_n\ \texttt{in}\ e_2\ \texttt{end end},\ \rho)$ when $n > 1$
8. $\mathit{type}(\texttt{local fun}\ f\ \texttt{(}x\ \texttt{:}\ s\texttt{) =}\ e_1\ \texttt{in}\ e_2\ \texttt{end},\ \rho) =$
   $\mathit{type}(\texttt{local var}\ f\ \texttt{= fn (}x\ \texttt{:}\ s\texttt{) =>}\ e_1\ \texttt{end in}\ e_2\ \texttt{end},\ \rho)$
9. $\mathit{type}(\texttt{local var}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end},\ \rho) = t_2$ if
   $\mathit{type}(e_1,\ \rho) = t_1$ for some t_1 and $\mathit{type}(e_2,\ \rho[x \mapsto t_1]) = t_2$
10. $\mathit{type}(\texttt{local fun rec}\ f\ \texttt{(}x\ \texttt{:}\ s\texttt{) :}\ t_1\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end},\ \rho) = t_2$ if
    $\mathit{type}(e_1,\ \rho[f \mapsto s\ \texttt{->}\ t_1][x \mapsto s]) = t_1$ and $\mathit{type}(e_2,\ \rho[f \mapsto s\ \texttt{->}\ t_1]) = t_2$
11. $\mathit{type}(\texttt{fn (}x\ \texttt{:}\ s\texttt{) :}\ t\ \texttt{=>}\ e\ \texttt{end},\ \rho) = s\ \texttt{->}\ t$ if $\mathit{type}(e,\ \rho[x \mapsto s]) = t$
12. $\mathit{type}(e_2\ e_1,\ \rho) = t_2$ if $\mathit{type}(e_2,\ \rho) = t_1\ \texttt{->}\ t_2$ for some t_1 and $\mathit{type}(e_1,\ \rho) = t_1$
13. $\mathit{type}(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\ \rho) = t$ if $\mathit{type}(e,\ \rho) = \texttt{bool}$ and $\mathit{type}(e_1,\ \rho) = \mathit{type}(e_2,\ \rho) = t$
14. $\mathit{type}(\texttt{not}\ e,\ \rho) = \texttt{bool}$ if $\mathit{type}(e,\ \rho) = \texttt{bool}$
15. $\mathit{type}(e_1\ \texttt{::}\ e_2,\ \rho) = t\ \texttt{list}$ if $\mathit{type}(e_1,\ \rho) = t$ and $\mathit{type}(e_2,\ \rho) = t\ \texttt{list}$
16. $\mathit{type}(\texttt{hd}\ e,\ \rho) = t$ if $\mathit{type}(e,\ \rho) = t\ \texttt{list}$
17. $\mathit{type}(\texttt{tl}\ e,\ \rho) = t\ \texttt{list}$ if $\mathit{type}(e,\ \rho) = t\ \texttt{list}$
18. $\mathit{type}(\texttt{ise}\ e,\ \rho) = \texttt{bool}$ if $\mathit{type}(e,\ \rho) = t\ \texttt{list}$
19. $\mathit{type}(\texttt{print}\ e,\ \rho) = \texttt{unit}$
20. $\mathit{type}(e_1\ \mathit{op}\ e_2,\ \rho) = \texttt{int}$ if $\mathit{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$ and $\mathit{type}(e_1,\ \rho) = \mathit{type}(e_2,\ \rho) = \texttt{int}$

21. $\textit{type}(e_1 \ \textit{op} \ e_2, \ \rho) = \texttt{bool}$ if $\textit{op} \in \{\texttt{<}, \texttt{<=}\}$ and $\textit{type}(e_1, \ \rho) = \textit{type}(e_2, \ \rho) = \texttt{int}$

22. $\textit{type}(e_1 \ \textit{op} \ e_2, \ \rho) = \texttt{bool}$ if $\textit{op} \in \{\texttt{=}, \texttt{<>}\}$ and $\textit{type}(e_1, \ \rho) = \textit{type}(e_2, \ \rho) = t$ for some **equality type** t

23. $\textit{type}(e_1 \ \texttt{;} \ e_2, \ \rho) = t$ if $\textit{type}(e_2, \ \rho) = t$