```
signature INTERPRETER=
   sig
      val interpret: string -> string
      val eval: bool ref
      and tc  : bool ref
   end;
```

The syntax of the language is as follows

exp	::=	*exp* `+` *exp*
		exp `-` *exp*
		exp `*` *exp*
		`true`
		`false`
		exp `=` *exp*
		`if` *exp* `then` *exp* `else` *exp*
		exp `::` *exp*
		`[` exp_1 `,` $\cdots$ `,` exp_n `]` $(n \geq 0)$
		`let` x `=` *exp* `in` *exp*
		`let rec` x `=` *exp* `in` *exp*
		x
		`fn` x `=>` *exp*
		exp `(` *exp* `)` (function application)
		n (natural numbers)
		`(` *exp* `)`

The abstract syntax of Mini ML is defined as a datatype in the signature `EXPRESSION`.

Exercise 1 Find this signature. What is the constructor corresponding to `let` expressions?

We program with signatures and functors only. After the signatures, which we shall not yet study, the first functor is the interpreter itself.

Exercise 2 Find this functor. Find the application of `Ty.prType`. Find it's type. What do you think `Ty.prType` is supposed to do? What is the type of `abstsyn`? What do you think the evaluator is supposed to do when asked to evaluate something which has not yet been implemented?

We shall now describe Version 1, the bare typechecker, and then proceed to the extensions.

4.1 VERSION 1: The bare Typechecker (Appendix A)

The first version is just able to type check integer constants and `+`. As signature `TYPE` reveals, the type `Type` of types is abstract, but there are functions we can use to build basic types and decompose them. `unTypeInt` is one of the latter; it is supposed to raise `Type` if applied to any Mini ML type different from the int (however the type int is represented). This is a common way of hiding implementation details, and it might be helpfull to look at how `functor Type` produces a structure which matches the signature `Type`.

As revealed by `signature TYPECHECKER`, the typechecker is going to depend on the abstract syntax and a `Type` structure. However, as you can see from the declaration of `functor TypeChecker`, all the typechecker knows about the implementation of types is what is specified by the signature `TYPE`. This allows us to experiment with the implementation of types to obtain greater efficiency without changing the typechecker, as we shall see in the later stages. As you see from `functor TypeChecker`, all the typechecker is capable of handling is integer constants and `+`.

Exercise 3 Modify the typechecker to handle `true`, `false`, and multiplication of integers.

Given the signature and functor declarations in Appendix A, one can build the system. First we import the parser

```
use "parser.sml";
```

and then we build the system by the following declarations (which can be read from file `build1.sml`).

```
structure Expression= Expression();

structure Parser= Parser(Expression);

structure Value = Value();

structure Evaluator=
   Evaluator(structure Expression= Expression
             structure Value = Value);

structure Ty = Type();

structure TyCh=
  TypeChecker(structure Ex = Expression
              structure Ty = Ty);

structure Interpreter=
  Interpreter(structure Ty= Ty
```

```
            structure Value = Value
            structure Parser = Parser
            structure TyCh = TyCh
            structure Evaluator = Evaluator);

open Interpreter;
```

4.2 VERSION 2: Adding lists and polymorphism

The first extension is to implement the type checking of lists. In Version 1 the type of an expression could be inferred either directly (as in the case of `true` and `false`, or from the type of the subexpressions (as in the case of the arithmetic operations). When we introduce list, this is no longer the case. Consider for example the expression

```
if ([] = [9]) then 5 else 7
```

Suppose we want to type check `([] = [9])` by first type checking the left subexpression `[]`, then the right subexpression `[9]` and finally checking that the left and right-hand sides are of the same type before returning the type `bool`. The problem now is that when we try to type check `[]` we cannot know that this empty list is supposed to be an integer list. The typechecker therefore just ascribes the type `'a list` to `[]`, where `'a` is a TYPE VARIABLE. The `[9]` of course turns out to be an `int list`. The typechecker now "compares" the two types `'a list` and `int list` and discovers that they can be made the same by applying the substitution that maps `'a` to `int`. Hence the type of the expression `[]` depends not just on the expression itself, but also on the context of the expression. The context can force the type inferred for the expression to become more specific.

This "comparison" of types performed by the typechecker is called UNIFICATION and is an algebraic operation of great importance in symbolic computing. Indeed, whole programming languages have evolved around the idea of unification (PROLOG, for example). Here is a couple of examples to illustrate how unifications works in the special case of interest, that of unifying types.

```
[ [] , [[5]] ]                                    (1)
```

This expression is well-typed! The point is that the `[]` can be regarded as an `int list list`. Let us see how the typechecker manages to infer the type `int list list list` for (1). The typechecker first rewrites the expression to the equivalent:

```
[] ::  (((5 ::  []) ::  []) ::  [])               (2)
```

Checking the first argument of the topmost `::` yields:

```
[] :  'a1 list                                    (3)
```

To check `(((5 :: []) :: []) :: [])`, we first check the left-hand `((5 :: []) :: [])`. To check this, we first check the left-hand `(5 :: [])`. To check this, we first check the left-hand `5`, for which the typechecker wisely infer the type `int`. Continuing to the right-hand part of `(5 :: [])`, `[]` gets the type `'a2 list`. To check the `::` of `(5 :: [])`, we now unify `int list` and `'a2 list`, which results in the substitution

$$S_1(\texttt{'a2}) = \texttt{int}.$$

Thus the type of `(5 :: [])` is `int list`.

Returning to `((5 :: []) :: [])`, the right-hand `[]` first gets type `'a3 list` which by unification with `int list list` yields the substitution

$$S_2(\texttt{'a3}) = \texttt{int list}.$$

Thus the type of `((5 :: []) :: [])` is `int list list`.

Returning to `(((5 :: []) :: []) :: [])`, the right-hand `[]` gets the type `'a4 list` which by unification with `int list list list` yields the substitution

$$S_3(\texttt{'a4}) = \texttt{int list list}$$

Thus the type of `(((5 :: []) :: []) :: [])` is `int list list list`.

Finally, returning to (2) and (3), we get to unify `'a1 list` with `int list list list`, yielding the substitution

$$S_4(\texttt{'a1}) = \texttt{int list list}.$$

The type of (2), and therefore the type of (1), is thus found to be `int list list list`.

Note that

```
[ [4] , [[5]] ]
```

is NOT well-typed. In an attempt to compute S_4, we would now be unifying `int list list` and `int list list list` and that gives a unification error.

To implement all this, we first extend the `TYPE` signature and introduce a new signature, `UNIFY`:

```
signature TYPE =
   sig
      eqtype tyvar
      val freshTyvar: unit -> tyvar
      ...
      val mkTypeTyvar: tyvar -> Type
          and unTypeTyvar: Type -> tyvar

      val mkTypeList: Type -> Type
          and unTypeList: Type -> Type
```

```
      type subst
      val Id: subst
                (* the identify substitution;   *)
      val mkSubst: tyvar*Type -> subst
                (* make singleton substitution; *)
      val on : subst * Type -> Type
                (* application;                  *)

      val prType: Type->string          (* printing *)
   end


signature UNIFY=
   sig
      structure Type: TYPE
      exception NotImplemented of string
      exception Unify
      val unify: Type.Type * Type.Type -> Type.subst
   end;
```

The nice thing is that we can extend the typechecker without knowing anything about the inner workings of unification, simply by including a formal parameter of signature `UNIFY` in the typechecker functor:

```
functor TypeChecker
  (...
   structure Ty: TYPE
   structure Unify: UNIFY
    sharing Unify.Type = Ty
  )=
struct
  infix on
  val (op on) = Ty.on
  ...

  fun tc (exp: Ex.Expression): Ty.Type =
   (case exp of
      ...
    | Ex.LISTexpr [] =>
        let val new = Ty.freshTyvar ()
         in Ty.mkTypeList(Ty.mkTypeTyvar  new)
        end
    | Ex.CONSexpr(e1,e2) =>
```