4 Implementing an Interpreter in ML

The purpose of this lecture is to show a worked example of program development using ML modules. We shall tackle the problem of implementing a small ML system. The system is of course going to considerable simplified compared to a real ML implementation.

We implement only a few of the language constructs found in real ML. The user of our system will not get the ability to declare new types and data types; however, there will be arithmetic on the build-in integers, `if`...`then`...`else` expressions, and indeed lists, higher order functions and recursion, so it is far from a trivial language. We shall refer to this language as Mini ML.

Moreover, the system will be an interpreter rather than a compiler. It still has a type-checker, indeed we shall see how one can implement a restricted form of polymorphism.

The system is actually running and you can modify and extend it provided you have access to an implementation and to the files listed in Appendix B. To make life easier for you, we provide a parse functor which can parse a string (the Mini ML source expression) into an ABSTRACT SYNTAX TREE, the shape of which will be defined below. The rest of the interpreter works on abstract syntax trees.

The interpreter uses a TYPECHECKER to check the validity of input expressions and an EVALUATOR to evaluate them. Initially, the typechecker and evaluator handle only a tiny subset of Mini ML. In this lecture I shall show how one in successive steps can extend the typechecker to handle polymorphic lists, variables and `let` expressions. In the practical sessions you can extend the evaluator in the same manner (it is easier than extending the typechecker).

The typechecker and the evaluator can be developed independently as long as you do not change the signatures we provide. This will allow you to take the typechecker functors I have written and plug into your own system as you improve the power of your evaluator. Alternatively, you might want to modify or extend my typechecker functors, and take over evaluator functors that other people write.

The source of the bare interpreter is in Appendix A. An overview of how to run the systems is provided in Appendix B.

The development of the typechecker and the evaluator need not be in step. You can disable either by assigning false to one of the variables `tc` and `eval`.

```
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) =>
```

```
        let val t1 = tc e1
            val t2 = tc e2
            val new = Ty.freshTyvar ()
            val newt= Ty.mkTypeTyvar new
            val t2' = Ty.mkTypeList newt
            val S1 = Unify.unify(t2, t2')
                     handle Unify.Unify=>
                     raise TypeError(e2,"expected list type")

            val S2 = Unify.unify(S1 on newt,S1 on t1)
                     handle Unify.Unify=>
                     raise TypeError(exp,
                      "element and list have different types")
         in S2 on (S1 on t2)
        end
    | ...

   )handle Unify.NotImplemented msg => raise NotImplemented msg

end; (*TypeChecker*)
```

We also have to extend the `Type` functor to meet the enriched `TYPE signature`. The easiest way of doing this is

```
functor Type():TYPE =
struct
  type tyvar = int
  val freshTyvar =
      let val r= ref 0 in fn()=>(r:= !r +1; !r) end
  datatype Type = INT
                | BOOL
                | LIST of Type
                | TYVAR of tyvar
  ...

  fun mkTypeTyvar tv = TYVAR tv
  and unTypeTyvar(TYVAR tv) = tv
    | unTypeTyvar _ = raise Type

  fun mkTypeList(t)=LIST t
  and unTypeList(LIST t)= t
    | unTypeList(_)= raise Type
```

```
  type subst = Type -> Type

  fun Id x = x

  fun mkSubst(tv,ty)=
     let fun su(TYVAR tv')= if tv=tv' then ty else TYVAR tv'
         |   su(INT) = INT
         |   su(BOOL)= BOOL
         |   su(LIST ty') = LIST (su ty')
      in su
     end

  fun on(S,t)= S(t)

  fun prType ...
  |   prType (LIST ty) = "(" ^ prType ty ^ ")list"
  |   prType (TYVAR tv) = "a" ^ makestring tv
end;
```

Exercise 4 Extend Version 2 to handle equality. All you have to do is to fill in the relevant case in the definition of the function `tc`. (See appendix B about how you get the source of Version 2).

4.3 VERSION 3: A different implementation of types

Version 3 arises from Version 2 by replacing the `Type` functor by a different implementation of types. The idea is that istead of having substitutions as functions, we can implement type variables by references (pointers) and then do substitutions directly by assignments.

In case you have not seen the reserved word `withtype` before, `withtype` is used to declare a type abbreviaton locally within a `datatype` declaration.

```
functor ImpType():TYPE =
struct
  datatype 'a option = NONE | SOME of 'a

  datatype Type = INT
                | BOOL
                | LIST of Type
                | TYVAR of tyvar

  withtype tyvar =  Type option ref
```

```
  type tyvar = Type option ref

  fun freshTyvar() = ref (NONE)

  exception Type

  fun mkTypeInt() = INT
  and unTypeInt(INT)=()
    | ...
    | unTypeInt(TYVAR(ref (SOME t)))= unTypeInt t
    | unTypeInt _ = raise Type


  ...
  type subst = unit

  val Id = ();

  exception MkSubst;

  fun mkSubst(tv,ty)=
     case tv of
       ref(NONE) => tv:= (SOME ty)
     | ref(SOME t) => raise MkSubst

  fun on(S,t)= t

  fun prType ...
  |   prType (TYVAR (ref NONE)) = "a?"
  |   prType (TYVAR (ref (SOME t))) = prType t
end;
```

We can now build two systems at the same time and compare the efficiency of the two implementations. The nice thing is that we do not have to modify the typechecker functor at all, nor do we even have to modify the unification functor; we can just extend the final sequence of structure declarations to use both implementations of types.

Exercise 5 When I did this, I found (to my surprise), that the functional version in some cases was twice as fast, and never slower than the imperative variant. The relative performance of the two vary greatly from expression to expression. Can you find an expression for which the imperative version really is faster? (See Appendix B for how to get hold of the source of Version 3). Be careful with generating very demanding tasks for the ML system; you can make it crash!

ML implementors normally opt for the imperative version. In all fairness, the above comparison ignores that composing substitutions is much easier in the imperative version than it is in the applicative version; in the fragment of Mini ML considered so far, we have not had to compose substitutions.

One should not be too concerned with performance issues at too early a stage. It can be surprisingly difficult to predict where efficiency is most needed, and it is much more important, at first, to get the overall structure of the system right. It was important, for example, that we did NOT make the constructors of the datatype `Type` visible in the signature `TYPE`, and that we wrote the unification algorithm in a way which does not use the internal structure of `Type`. Had we not done this, we would not have been able to switch from one implementation to another that easily, and therefore chances are that we would chosen the imperative one, assuming that it was the more efficient one, without ever trying the "obvious" applicative implentation.

4.4 VERSION 4: Introducing variables and `let`

We now extend Version 3 by implementing the type checking of `let` expressions and of identifiers.

The typechecker function `tc` now has to take TWO arguments,

```
tc(TE, e)
```

where `e` is an expression and `TE` is a TYPE ENVIRONMENT, which maps variables occurring free in `e` to TYPE SHEMES. The definition of what a type scheme is will be given below; for now it suffices to know that every type can be regarded as a type scheme.

To take an example, if `TE` maps `x` to `int` and `y` to `int`, then `tc` will deduce the type `int` for the expression `x+y`. (However, if `TE` mapped `y` to `bool`, there would be a type error.)

The fact that we can bind variables to expressions whose types have been inferred to contain type variables means that we get type variables in the type environment. For instance, to type check

```
let x = [] in 4 :: x end
```

we first check `[]` yielding the type `'a1 list`, say. Then we bind `x` to the type scheme $\forall$`'a1.'a1 list`. Here the binding $\forall$`'a1` of `'a1` indicates that when we look up the the type of `x` in the type environment, we return a type obtained from the type scheme $\forall$`'a1.'a1 list` by instantiating the bound variables (here just `'a1`) by fresh type variables. In our example, when we look up `x` in the type environment during the checking of `4 :: x`, we instantiate `'a1` to a fresh type variable `'a2`, say, yielding the type `'a2 list` for `x`. Thus we get to unify `int list` against `'a2 list`, yielding the substitution of `int` for `'a2`.

Throughout the body of the `let`, `x` will be bound to $\forall$`'a1.'a1 list` in the type environment. Since we take a fresh instance of this type scheme each time we look up `x`, we can use `x` both as an `int list` and as an `int list list`, say:

```
let x = [] in   (4::x)::x end
```

Exercise 6 Assuming that you instantiate the bound 'a1 to 'a3 when you meet the last occurrence of x, what two types should be unified, and what is the resulting substitution on 'a3 ?

The variable x is an example of POLYMORPHISM: after x has been declared, an occurrence of x can potentially be given infinitely many types: int list, bool list, int list list, and so on, all captured by the type scheme $\forall$ 'a1.'a1 list. In ML, a TYPE SCHEME always takes the form $\forall\, \alpha_1 \cdots \alpha_n.\tau$, $(n \geq 0)$, where $\alpha_1, \ldots, \alpha_n$ are type variables and τ is a type. In the fragment of Mini ML considered so far, it will always be the case that any type variable occurring in τ is amongst the $\alpha_1, \ldots, \alpha_n$, but when one introduces functions and application, this no longer is the case.

Here is how we implement variables and let. We first extend the TYPE signature:

```
signature TYPE =
   sig
      ...
      type TypeScheme

      val instance: TypeScheme -> Type
      val close: Type -> TypeScheme

   end
```

Version 1 (Appendix A) already contains a signature for environments (find it). It was actually intended for the practical where you need it to extend the evaluator, but we can make use of it to implement type environments. The signature of the typechecker can be left unchanged, but we need to change the functor that builds the typechecker by including the environment management among the formal parameters:

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

```
exception NotImplemented of string
exception TypeError of Ex.Expression * string

fun tc (TE: Ty.TypeScheme TE.Environment, exp: Ex.Expression): Ty.Type =
 (case exp of
    Ex.BOOLexpr b => Ty.mkTypeBool()
  | Ex.NUMBERexpr _ => Ty.mkTypeInt()
  | Ex.SUMexpr(e1,e2)  => checkIntBin(TE,e1,e2)
  | Ex.DIFFexpr(e1,e2) => checkIntBin(TE,e1,e2)
  | Ex.PRODexpr(e1,e2) => checkIntBin(TE,e1,e2)
  | Ex.LISTexpr [] =>
       let val new = Ty.freshTyvar ()
        in Ty.mkTypeList(Ty.mkTypeTyvar  new)
       end
  | Ex.LISTexpr(e::es) => tc (TE, Ex.CONSexpr(e,Ex.LISTexpr es))
  | Ex.CONSexpr(e1,e2) =>
      let val t1 = tc(TE, e1)
          val t2 = tc(TE, e2)
          val new = Ty.freshTyvar ()
          val newt= Ty.mkTypeTyvar new
          val t2' = Ty.mkTypeList newt
          val S1 = Unify.unify(t2, t2')
                   handle Unify.Unify=>
                   raise TypeError(e2,"expected list type")

          val S2 = Unify.unify(S1 on newt,S1 on t1)
                   handle Unify.Unify=>
                   raise TypeError(exp,"element and list have different types")
       in S2 on (S1 on t2)
      end
  | Ex.EQexpr _ => raise NotImplemented "(equality)"
  | Ex.CONDexpr _ => raise NotImplemented "(conditional)"
  | Ex.DECLexpr(x,e1,e2) =>
       let val t1 = tc(TE,e1);
           val typeScheme = Ty.close(t1)
        in tc(TE.declare(x,typeScheme,TE), e2)
       end
  | Ex.RECDECLexpr _ => raise NotImplemented "(rec decl)"
  | Ex.IDENTexpr x   =>
       (Ty.instance(TE.retrieve(x,TE))
       handle TE.Retrieve _ =>
        raise TypeError(exp,"identifier " ^ x ^ " not declared"))
  | Ex.LAMBDAexpr _  => raise NotImplemented "(function)"
```

```
    | Ex.APPLexpr _ => raise NotImplemented    "(application)"

   )handle Unify.NotImplemented msg => raise NotImplemented msg

  and checkIntBin(TE,e1,e2) =
    let val t1 = tc(TE,e1)
        val _  = Ty.unTypeInt t1
                 handle Ty.Type=> raise TypeError(e1,"expected int")
        val t2 = tc(TE,e2)
        val _  = Ty.unTypeInt t2
                 handle Ty.Type=> raise TypeError(e2,"expected int")
     in Ty.mkTypeInt()
    end;

  fun typecheck(e) = tc(TE.emptyEnv,e)

end; (*TypeChecker*)
```

Then we extend the Type functor to match the TYPE signature:

```
functor Type():TYPE =
struct
  ...
  datatype TypeScheme = FORALL of tyvar list * Type

  fun instance(FORALL(tyvars,ty))=
  let val old_to_new_tyvars = map (fn tv=>(tv,freshTyvar())) tyvars
      exception Find;
      fun find(tv,[])= raise Find
      |   find(tv,(tv',new_tv)::rest)=
          if tv=tv' then new_tv else find(tv,rest)
      fun ty_instance INT = INT
      |   ty_instance BOOL = BOOL
      |   ty_instance (LIST t) = LIST(ty_instance t)
      |   ty_instance (TYVAR tv) =
             TYVAR(find(tv,old_to_new_tyvars)
                   handle Find=> tv)

  in
     ty_instance ty
  end
```

```
  fun close(ty)=
  let fun fv(INT) = []
      |   fv(BOOL)= []
      |   fv(LIST t) = fv(t)
      |   fv(TYVAR tv) = [tv]
   in FORALL(fv ty,ty)
  end

end;
```

Finally, the system is re-built as in Version 2, except that we have to provide and link in an `Environment` functor which matches `ENVIRONMENT`.

Exercise 7 Extend Version 4 with `if .. then .. else`. (This extension has no subtle implications for the type checking.)

Exercise 8 [For the extra keen] Extend Version 4 to cope with lambda abstraction (`fn`) and application. First, you have to introduce arrow types with constructors and destructors. Then you have to change the type of `close` so that it takes two arguments, namely a type environment and a type. It should return the type scheme that is obtained by quantifying on all the type variables that occur in the type but do not occur free in the type environment.

Then you can modify the type checker. When you type check a lambda abstraction, you just bind the formal parameter to the trivial type scheme which is just a fresh type variable (no quantified variables). Thus the type environment can now contain type schemes with free type variables.

An application `tc(TE,e)` now yields two arguments, namely a type t and a substitution S; the idea is that if you apply the substitution S to the type environment `TE`, which now can contain free type variables, the expression `e` has the type t. When an expression consists of more than one subexpression, the type environment gradually becomes more and more specific by applying the substitutions produced by the checking of the subexpressions one by one. Moreover, the substitution returned from the whole expression is the composition of these individual substitutions. (You have to extend the `TYPE` signature (and the `Type` functor) with composition of substitutions.

Finally, you can extend the unification algorithm to cope with arrow types. (This will also use composition of substitutions.)

4.5 Acknowledgement

The parser and evaluator and all the signatures related to them are due to Nick Rothwell.

Appendix A: The bare Interpreter

```
(* interp1.sml : VERSION 1: the bare interpreter *)

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

                  (* syntax *)

signature EXPRESSION =
   sig
      datatype Expression =
         SUMexpr of Expression * Expression    |
         DIFFexpr of Expression * Expression   |
         PRODexpr of Expression * Expression   |
         BOOLexpr of bool   |
         EQexpr of Expression * Expression   |
         CONDexpr of Expression * Expression * Expression   |
         CONSexpr of Expression * Expression   |
         LISTexpr of Expression list   |
         DECLexpr of string * Expression * Expression   |
         RECDECLexpr of string * Expression * Expression   |
         IDENTexpr of string   |
         LAMBDAexpr of string * Expression   |
         APPLexpr of Expression * Expression   |
         NUMBERexpr of int
   end


              (* parsing *)

signature PARSER =
   sig
      structure E: EXPRESSION

      exception Lexical of string
      exception Syntax of string
```

```
      val parse: string -> E.Expression
   end


                        (* environments *)

signature ENVIRONMENT =
   sig
      type 'object Environment

      exception Retrieve of string

      val emptyEnv: 'object Environment
      val declare: string * 'object * 'object Environment
          -> 'object Environment
      val retrieve: string * 'object Environment -> 'object
   end

                        (* evaluation *)
signature VALUE =
   sig
      type Value
      exception Value

      val mkValueNumber: int -> Value
          and unValueNumber: Value -> int

      val mkValueBool: bool -> Value
          and unValueBool: Value -> bool

      val ValueNil: Value
      val mkValueCons: Value * Value -> Value
          and unValueHead: Value -> Value
          and unValueTail: Value -> Value

      val eqValue: Value * Value -> bool
      val printValue: Value -> string
   end


signature EVALUATOR =
   sig
```

```
      structure Exp: EXPRESSION
      structure Val: VALUE
      exception Unimplemented
      val evaluate: Exp.Expression -> Val.Value
   end

                  (* type checking *)
signature TYPE =
   sig
      type Type

(*constructors and decstructors*)
      exception Type
      val mkTypeInt: unit -> Type
          and unTypeInt: Type -> unit

      val mkTypeBool: unit -> Type
          and unTypeBool: Type -> unit

      val prType: Type->string
   end



signature TYPECHECKER =
   sig
      structure Exp: EXPRESSION
      structure Type: TYPE
      exception NotImplemented of string
      exception TypeError of Exp.Expression * string
      val typecheck: Exp.Expression -> Type.Type
   end;

                  (* the interpreter*)

functor Interpreter
   (structure Ty: TYPE
    structure Value : VALUE
    structure Parser: PARSER
    structure TyCh: TYPECHECKER
    structure Evaluator:EVALUATOR
      sharing Parser.E = TyCh.Exp = Evaluator.Exp
          and TyCh.Type = Ty
```

```
        and Evaluator.Val = Value
   ): INTERPRETER=

struct
  val eval= ref true    (* toggle for evaluation *)
  and tc  = ref true    (* toggle for type checking *)
  fun interpret(str)=
    let val abstsyn= Parser.parse str
        val typestr= if !tc then
                     Ty.prType(TyCh.typecheck abstsyn)
                     else "(disabled)"
        val valuestr= if !eval then
                  Value.printValue(Evaluator.evaluate abstsyn)
                      else "(disabled)"

    in  valuestr ^ " : " ^ typestr
    end
    handle Evaluator.Unimplemented =>
               "Evaluator not fully implemented"
         | TyCh.NotImplemented msg =>
               "Typechecker not fully implemented " ^ msg
         | Value.Value   => "Run-time error"
         | Parser.Syntax msg => "Syntax Error: " ^ msg
         | Parser.Lexical msg=> "Lexical Error: " ^ msg
         | TyCh.TypeError(_,msg)=> "Type Error: " ^ msg
end;

                    (* the evaluator *)

functor Evaluator
  (structure Expression: EXPRESSION
   structure Value: VALUE):EVALUATOR=

   struct
      structure Exp= Expression
      structure Val= Value
      exception Unimplemented

      local
         open Expression Value
         fun evaluate exp =
            case exp
              of BOOLexpr b => mkValueBool b
```

```
              | NUMBERexpr i => mkValueNumber i
              | SUMexpr(e1, e2) =>
                  let val e1' = evaluate e1
                      val e2' = evaluate e2
                  in
                     mkValueNumber(unValueNumber e1' +
                                   unValueNumber e2')
                  end

              | DIFFexpr(e1, e2) =>
                  let val e1' = evaluate e1
                      val e2' = evaluate e2
                  in
                     mkValueNumber(unValueNumber e1' -
                                   unValueNumber e2')
                  end

              | PRODexpr(e1, e2) =>
                  let val e1' = evaluate e1
                      val e2' = evaluate e2
                  in
                     mkValueNumber(unValueNumber e1' *
                                   unValueNumber e2')
                  end

              | EQexpr _ => raise Unimplemented
              | CONDexpr _ => raise Unimplemented
              | CONSexpr _ => raise Unimplemented
              | LISTexpr _ => raise Unimplemented
              | DECLexpr _ => raise Unimplemented
              | RECDECLexpr _ => raise Unimplemented
              | IDENTexpr _ => raise Unimplemented
              | LAMBDAexpr _ => raise Unimplemented
              | APPLexpr _ => raise Unimplemented

    in
       val evaluate = evaluate
    end
  end;

                         (* the typechecker *)

functor TypeChecker
```

```
  (structure Ex: EXPRESSION
   structure Ty: TYPE)=
struct
  structure Exp = Ex
  structure Type = Ty
  exception NotImplemented of string
  exception TypeError of Ex.Expression * string

  fun tc (exp: Ex.Expression): Ty.Type =
    case exp of
      Ex.BOOLexpr b => raise NotImplemented
                         "(boolean constants)"
    | Ex.NUMBERexpr _ => Ty.mkTypeInt()
    | Ex.SUMexpr(e1,e2)  => checkIntBin(e1,e2)
    | Ex.DIFFexpr _ => raise NotImplemented "(minus)"
    | Ex.PRODexpr _ => raise NotImplemented "(product)"
    | Ex.LISTexpr _ => raise NotImplemented "(lists)"
    | Ex.CONSexpr _ => raise NotImplemented "(lists)"
    | Ex.EQexpr _ => raise NotImplemented "(equality)"
    | Ex.CONDexpr _ => raise NotImplemented "(conditional)"
    | Ex.DECLexpr _ => raise NotImplemented "(declaration)"
    | Ex.RECDECLexpr _ => raise NotImplemented "(rec decl)"
    | Ex.IDENTexpr _   => raise NotImplemented "(identifier)"
    | Ex.LAMBDAexpr _  => raise NotImplemented "(function)"
    | Ex.APPLexpr _ => raise NotImplemented "(application)"

  and checkIntBin(e1,e2) =
    let val t1 = tc e1
        val _  = Ty.unTypeInt t1
                 handle Ty.Type=>
                 raise TypeError(e1,"expected int")
        val t2 = tc e2
        val _  = Ty.unTypeInt t2
                 handle Ty.Type=>
                 raise TypeError(e2,"expected int")
     in Ty.mkTypeInt()
    end;

  val typecheck = tc

end; (*TypeChecker*)
```

```
                    (* the basics -- nullary functors *)

functor Type():TYPE =
struct
  datatype Type = INT
                | BOOL

  exception Type

  fun mkTypeInt() = INT
  and unTypeInt(INT)=()
    | unTypeInt(_)= raise Type

  fun mkTypeBool() = BOOL
  and unTypeBool(BOOL)=()
    | unTypeBool(_)= raise Type

  fun prType INT = "int"
  |   prType BOOL= "bool"
end;



functor Expression(): EXPRESSION =
   struct
      type 'a pair = 'a * 'a

      datatype Expression =
         SUMexpr of Expression pair    |
         DIFFexpr of Expression pair    |
         PRODexpr of Expression pair    |
         BOOLexpr of bool    |
         EQexpr of Expression pair    |
         CONDexpr of Expression * Expression * Expression    |
         CONSexpr of Expression pair    |
         LISTexpr of Expression list    |
         DECLexpr of string * Expression * Expression    |
         RECDECLexpr of string * Expression * Expression    |
         IDENTexpr of string    |
         LAMBDAexpr of string * Expression    |
         APPLexpr of Expression * Expression    |
         NUMBERexpr of int
```

```
   end;

functor Value(): VALUE =
   struct
      type 'a pair = 'a * 'a

      datatype Value = NUMBERvalue of int   |
                       BOOLvalue of bool   |
                       NILvalue   |
                       CONSvalue of Value pair

      exception Value

      val mkValueNumber = NUMBERvalue
      val mkValueBool = BOOLvalue

      val ValueNil = NILvalue
      val mkValueCons = CONSvalue

      fun unValueNumber(NUMBERvalue(i)) = i   |
          unValueNumber(_) = raise Value

      fun unValueBool(BOOLvalue(b)) = b   |
          unValueBool(_) = raise Value

      fun unValueHead(CONSvalue(c, _)) = c   |
          unValueHead(_) = raise Value

      fun unValueTail(CONSvalue(_, c)) = c   |
          unValueTail(_) = raise Value

      fun eqValue(c1, c2) = (c1 = c2)

(* Pretty-printing *)
      fun printValue(NUMBERvalue(i)) = makestring(i)   |
          printValue(BOOLvalue(true)) = "true"   |
          printValue(BOOLvalue(false)) = "false"   |
          printValue(NILvalue) = "[]"   |
          printValue(CONSvalue(cons)) = "[" ^
                 printValueList(cons) ^ "]"
          and printValueList(hd, NILvalue) = printValue(hd) |
              printValueList(hd, CONSvalue(tl)) =
                 printValue(hd) ^ ", " ^ printValueList(tl) |
```

```
            printValueList(_) = raise Value
end;
```

Appendix B: Files

The following files are available in the directory `/usr/cheops/mads/course`

- `interp1.sml` Version 1 (as included in Appendix A).
- `interp2.sml` ⋯ `interp4.sml` The other versions.
- `build1.sml` the structure declarations needed to build Version 1.
- `build2.sml` ⋯ `build4.sml` Similarly for the other versions.
- `parser.sml` The parser functor.

To build Version 3, say, you type the following (assuming you have copied the files to your directory):

```
use "interp3.sml";
use "parser.sml";
use "build3.sml";
```

Since the parser functor is completely closed, you dont have to include it more than once in every session, although you will probably want to build your system several times while you experiment with the extensions.