A Simple Interpreter in ML

- Goal is to experiment with implementation issues in a language interpreter
- Assumptions:
  - The language is expression based (like ML and Scheme)
  - We already have an abstract syntax tree
  - The interpreter will interpret (evaluate) the tree
- Implementation will be in ML (but we could do the same in Scheme)

Expression Grammar in ML

- Start by designing a simple grammar for expressions
- What should the type look like in ML?
  - Simplest expressions would be numbers (integers)
  - Expressions could be binary expressions (e.g., sums or products of integers)
  - Grammar rule would be
    \[ \texttt{<expr>} \rightarrow \texttt{NUM} \mid \texttt{<expr>} + \texttt{<expr>} \]
- First attempt: define a data type in ML
  \[
  \text{datatype Exp} = \text{int} \mid \text{Exp} * \text{Exp}
  \]
- What's wrong with this?
Expression Type

- Think of an expression as being built from either an integer or by adding two expressions.
- Use ML type constructors:
  
  ```ml
  datatype Exp = Num of int | Plus of Exp * Exp;
  ```
- Thus the Exp type is a unifying type and can be constructed from two patterns.
- Examples of Exp values:
  
  ```ml
  val v1 = Num(5);
  val v1 = Num 5 : Exp
  val v2 = Plus(Num(4), Num(7));
  val v2 = Plus (Num 4, Num 7) : Exp
  val v3 = Plus(Num(4), Plus(Num(7), Num(9)));
  val v3 = Plus (Num 4, Plus (Num #, Num #)) : Exp
  ```

A Very Simple Interpreter

- Values of type Exp are like syntax trees of an expression.
- Now we can write a function to evaluate the expression:
  
  ```ml
  fun Interp(Num n) = n
  |
  Interp(Plus(e1,e2)) = Interp(e1) + Interp(e2);
  ```
- Examples of Exp values:
  
  ```ml
  Interp v1;
  val it = 5 : int
  Interp v2;
  val it = 11 : int
  Interp v3;
  val it = 20 : int
  ```
Extending Arithmetic

- Implement integer division in the interpreter
- Change the data type to include quotient expressions

```
datatype Exp = Num of int | Plus of Exp * Exp
            | Div of Exp * Exp;
```

- Add a pattern to Interp
  - fun Interp(Num(n)) = n
  - | Interp(Plus(e1, e2)) = Interp(e1) + Interp(e2)
  - | Interp(Div(e1, e2)) = Interp(e1) div Interp(e2);

```
val Interp = fn : Exp -> int
```

- Example
  - Interp(Div(Num(20), Num(4)));

```
val it = 5 : int
```

Handling Errors

- Check for division by zero
  - fun Interp(Num(n)) = n
  - | Interp(Plus(e1, e2)) = Interp(e1) + Interp(e2)
  - | Interp(Div(e1, Num(0))) = "Divide by zero!"
  - | Interp(Div(e1, e2)) = Interp(e1) div Interp(e2);

```
Error: right-hand-side of clause doesn't agree with
       function result type,
       expression:string, result type:int
```

- What's wrong with this?
  - We want to handle the error, but we must always return a value
    for the expression. But there is no valid integer value for divide
    by zero, so what should we return?
Exceptions

- ML has exceptions
- Exceptions can be declared as special types
- Exceptions can be raised
- Exceptions can be caught

Example of use:
- exception DivideByZero;
- fun Interp(Num(n)) = n
  = | Interp(Plus(e1, e2)) = Interp(e1) + Interp(e2)
  = | Interp(Div(e1, Num(0))) = raise DivideByZero
  = | Interp(Div(e1, e2)) = Interp(e1) div Interp(e2);
val Interp = fn : Exp -> int

- Interp(Div(Num(3), Num(0)));
  uncaught exception DivideByZero

Another Approach

- Define an error type as another kind of return value

datatype Op = PLUS | DIV;
datatype Exp = Num of int | Exp of Op * Exp * Exp;
datatype Retval = Int of int | Error;

fun Interp (Num(n)) = Int(n)
  | Interp (Exp(oper, el, e2)) = 
    let val (x,y) = (Interp(el),Interp(e2))
    in case (x,y) of
      (Int(n1),Int(n2)) => ( case oper of
        PLUS => Int(n1 + n2)
        | DIV => if (n2 <> 0) then Int(n1 div n2)
          else Error )
      | (_,_) => Error
    end
  ;
Adding Variables

- Define an environment as a list of variables with values
  
  ```
  datatype Env = Vlist of (string * Result) list
  and Result = Int of int
  ```

- Add an expression type of variable
  
  ```
  datatype Exp = Variable of string | Num of int |...
  ```

- Evaluate interpreter with an environment
  
  ```
  val env = Vlist ["x", Int(7)];
  val env = Vlist [("x",Int 7)] : Env
  - interp1( Plus(Variable("x"), Num(9)), env);
  val it = Int 16 : Result
  ```

Adding Variables

- Implementation of interpreter with an environment
  
  ```
  fun interp1(exp,env) =
      case exp of
          Variable(id) => lookup(env,id)
      | Num(n) => Int(n)
      | Plus(e1, e2) =>
          let val (v1,v2)=(interp1(e1,env),interp1(e2,env))
          in case (v1,v2) of
              (Int(n), Int(m)) => Int(n+m)
          | _ => raise Error("Bad operands")
      end
  ```
Adding Variable Binding

- Add a binding expression (like a Scheme or ML "let")
  
  ```
  datatype Exp = ... | Lett of ((string * Exp) list) * Exp
  ```

- Add a case for binding

  ```
  | Lett(id_e_list, exp) =>
  
  let val id_r_list =
      map (fn (id,e)()=>(id,interp1(e,env))) id_e_list
  in
      interp1(exp, extend_env_all(env, id_r_list))
  end
  ```

Bind a variable and use it in an expression

```latex
- val env = Vlist [("x", Int(7))];
  val env = Vlist [("x",Int 7)] : Env

  - interp1(
      Lett( [("y", Num(9))],
          Plus(Variable("x"), Variable("y")),
          env);
      val it = Int 16 : Result
  ```
Adding Functions as a Type

- Add a new expression type for functions (like a Scheme "lambda" or ML "fn"). For simplicity, allow one parameter.
  \[\text{datatype } \text{Exp} = \ldots | \text{Lambda of identifier } \ast \text{Exp}\]
  - parameter name
  - function body

- Add a case for evaluating a function (not a function call, just a function value, which is itself)
  \[| \text{Lambda(id, exp)} => \text{Function(id,exp)}\]

- Note this requires a new return value type
  \[\text{datatype } \text{Result} = \ldots | \text{Function of string } \ast \text{Exp}\]

Adding Function Application

- Add a new expression type for function for function application. Recall we allow one parameter. First expr is the function, second is the parameter value.
  \[\text{datatype } \text{Exp} = \ldots | \text{App of Exp } \ast \text{Exp}\]

- Add a case for interpreting function call
  \[| \text{App(e1, e2)} =>\]
  \[\begin{align*}
  \text{let val } (v1,v2) &= \text{(interp1(e1,env), interp1(e2,env))} \\
  \text{in case } v1 \text{ of} \\
  \text{Function(id,exp)} &= \text{interp1(exp,extend_env(env,id,v2))} \\
  | \_ &= \text{raise Error("Not a function")}
  \end{align*}\]

  - bind the parameter value to the parameter name in the env
Example of Function Application

- Bind foo to a function, then call it with value 7
  ```
  interp1(
    Lett( ["foo",
          Lambda("x", Plus(Variable("x"), Num(10)))],
          App(Variable("foo"), Num(7)), Vlist([]));
  val it = Int 17 : Result
  ```

- But what about scope? Environment at point of function definition should be saved for use in application to get static scope.

- This is called the closure (wrapping a function with the environment in effect at time of definition)