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
    \[
    \text{<expr>} \rightarrow \text{NUM} \mid \text{<expr>} + \text{<expr>}
    \]
- First attempt: define a data type in ML
  \[
  \text{datatype Exp} = \text{int} \mid \text{Exp * Exp}
  \]
- 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:
  - `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
  - `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 7, Num 9)) : 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
  - `fun Interp(Num n) = n`
  - `| Interp(Plus(e1, e2)) = Interp(e1) + Interp(e2);`
- Note the type of the Interp function
- Examples of Exp values
  - `val v1 = 5 : int`
  - `val v2 = 11 : int`
  - `val v3 = 20 : int`
### Extending Arithmetic

- Implement integer division in the interpreter
- Change the data type to include quotient expressions
  ```ml
datatype Exp = Num of int | Plus of Exp * Exp
  | Div of Exp * Exp;
  ```
- Add a pattern to `Interp`
  ```ml
  - 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
  ```ml
  - Interp(Div(Num(20), Num(4)));
  val it = 5 : int
  ```

### Handling Errors

- Check for division by zero
  ```ml
  - 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
  ```ml
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(opr, e1, e2)) =
  let val (x,y) = (Interp(e1),Interp(e2))
  in case (x,y) of
    (Int(n1),Int(n2)) => ( case op of
      PLUS => Int(n1 + n2)
    | DIV => if (n2 <> 0) then Int(n1 div n2) else Error )
    | (_,_) => Error
  end

```

---

Spring 2013  
CIS 425 - ML  

7

---

Spring 2013  
CIS 425 - ML  

8
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)
      end
Adding Variable Binding

- Add a binding expression (like a Scheme or ML "let")
  \[
  \text{datatype Exp } = \ldots | \text{Lett of ((string * Exp) list) * Exp}
  \]

- Add a case for binding
  \[
  | \text{Lett(id\_e\_list, exp) =>}
  \text{let val id\_r\_list =}
  \text{map (fn (id,e)=>(id,interp1(e,env))) id\_e\_list}
  \text{in}
  \text{interp1(exp, extend\_env\_all(env, id\_r\_list))}
  \text{end}
  \]

Bind a variable and use it in an expression

- \[\text{val env} = \text{Vlist ["x", Int(7)]};\]
- \[\text{val env} = \text{Vlist ["x", Int 7]} : \text{Env}\]

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

- Add a new expression type for functions (like a Scheme "lambda" or ML "fn"). For simplicity, allow one parameter.
  
  ```plaintext
datatype Exp = ...| Lambda of string * Exp
```

- Add a case for evaluating a function (not a function call, just a function value, which is itself)
  
  ```plaintext
| Lambda(id, exp) => Function(id,exp)
```

- Note this requires a new return value type
  
  ```plaintext
datatype Result = ... | Function of string * Exp
```

Adding Function Application

- Add a new expression type for function application. Recall we allow one parameter. First expr is the function, second is the parameter value.
  
  ```plaintext
datatype Exp = ...| App of Exp * Exp
```

- Add a case for interpreting function call
  
  ```plaintext
| App(e1, e2) =>

  let val (v1,v2) = (interp1(e1,env), interp1(e2,env))

  in case v1 of

  Function(id,exp) =>interp1(exp,extend_env((id,v2),env))

  | _ => raise Error("Not a function")

  end
```

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

```ml
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)

Adding Function Closure

- Add environment variable list to Function type

```ml
datatype Env = Vlist of (string * Result) list  
and Result = Int of int  
| Function of string * Exp * Env
```

- Store current environment with a function object

```ml
| Lambda(id, exp) => Function(id, exp, env)
```

- Evaluate function application body with saved environment

```ml
| App(e1, e2) =>  
  let val (v1,v2) = (interp1(e1,env), interp1(e2,env))  
  in case v1 of Function(id, exp, closure)  
    => interp1(exp, extend_env((id,v2), closure))
```