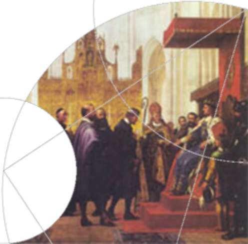
Interpretation (Continued)

Slides by Cosmin E. Oancea
cosmin.oancea@diku.dk

(with tweaks by Andrzej Filinski, andrzej@di.ku.dk)

Department of Computer Science (DIKU)
University of Copenhagen

May 2023 IPS Lecture Slides



Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree → Interpretation

↓

Typecheck

↓

Syntax tree

↓

Intermediate code generation ⟶ Intermediate code

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

↑

Assembly and linking

↑

Binary machine code

1. Intuition: Working with Abstract Syntax Trees and Symbol Tables
   - Basic expressions from last time
   - Interpreting Function Calls

2. Interpretation in textbook notation

Our abstract syntax so far

```
type Value = IntVal  of int
           | BoolVal of bool

type Exp = Constant of Value
         | Plus   of Exp * Exp
         | Minus  of Exp * Exp
         | Times  of Exp * Exp
         | Divide of Exp * Exp
         | Let    of string * Exp * Exp
         | Var    of string
         | If     of Exp * Exp * Exp
         | Less   of Exp * Exp
         | Equal  of Exp * Exp
         | And    of Exp * Exp
         | Or     of Exp * Exp
```

Our symbol tables

```
type SymTab<'a> = SymTab of (string * 'a) list

// empty : unit -> SymTab<'a>
let empty () = SymTab []

// bind : string -> 'a -> SymTab<'a> -> SymTab<'a>
let bind n i (SymTab stab) = SymTab ((n,i)::stab)

// lookup : string -> SymTab<'a> -> 'a option
let rec lookup n tab = match tab with
                         | SymTab [] -> None
                         | SymTab ((n1,i1)::remtab) ->
                           if n = n1 then Some i1
                           else lookup n (SymTab remtab)
```

Our evaluator

```
exception MyError of string

// eval : SymTab<Value> -> Exp -> Value
let rec eval vtable e =
  match e with
    | Constant v -> v
    | Plus (e1, e2) ->
      match (eval vtable e1, eval vtable e2) with
        | (IntVal n1, IntVal n2) -> IntVal (n1 + n2)
        | _ -> raise (MyError "Operands to + are not ints")
    ...
    | Var x ->
      match lookup x vtable with
        | None -> raise (MyError ("Unknown variable " + x))
        | Some v -> v
    | Let (x, e1, e2) ->
      let v1 = eval vtable e1
      let vtable1 = bind x v1 vtable
      eval vtable1 e2
    ...
```

1. Intuition: Working with Abstract Syntax Trees and Symbol Tables
   - Basic expressions from last time
   - Interpreting Function Calls
2. Interpretation in textbook notation

Interpreting Larger Languages

Let us add user-definable functions to our language of arithmetic expressions:

```
type Type  = Int | Bool
type Param = Type * string

type FunDec =
  FunDec of Type * string * Param list * Exp
```

Interpreting Larger Languages

Let us add user-definable functions to our language of arithmetic expressions:

```
type Type  = Int | Bool
type Param = Type * string

type FunDec =
  FunDec of Type * string * Param list * Exp
```

A function declaration consists of a return type, name, list of parameters, and a body. A parameter consists of a type and a name.

To add function calls, we add another constructor to Exp:

```
type Exp = ...
         | Call of string * Exp list
```

Interpreting Larger Languages

Let us add user-definable functions to our language of arithmetic expressions:

```
type Type  = Int | Bool
type Param = Type * string

type FunDec =
  FunDec of Type * string * Param list * Exp
```

A function declaration consists of a return type, name, list of parameters, and a body. A parameter consists of a type and a name.

To add function calls, we add another constructor to Exp:

```
type Exp = ...
         | Call of string * Exp list
```

the name of the function + a list of actual-argument expressions

Generalising the Symbol Table

We need two symbol tables: a *variable table*, and a *function table*.

The *function table* binds function names to their definitions (when interpreting).

Both map `strings` to information, so we parametrise on the type of information:

```
type VarTab = SymTab<Value>
type FunTab = SymTab<FunDec>
```

We do not have to modify the symbol table functions.

Function-Call Interpretation

We add another parameter, `ftable`, to the `eval` function. It is passed down recursively just like the `vtable`, so we will need to modify every case:

```
// eval : VarTab -> FunTab -> Exp -> Value
let rec eval vtable ftable e =
  match e with
    | Constant v -> v
    | Plus (e1, e2) ->
      match(eval vtable ftable e1, eval vtable ftable e2) with
        | (IntVal n1, IntVal n2) -> IntVal (n1 + n2)
        | _ -> raise (MyError "Operands to + are not ints")
    | ...
```

You get the picture.

Function-Call Interpretation

Recall we have added another constructor to `Exp`:

```
type Exp = ...
         | Call of string * Exp list
```

The interesting case is the new one for `Call`:

```
let rec eval vtable ftable e =
match e with
  ...
  | Call (fname, args) ->
    let vals = List.map(fun arg->eval vtable ftable arg) args
    match lookup fname ftable with
     | None -> raise (MyError ("Unknown function " + fname))
     | Some fundec -> callFun ftable fundec vals
```

Function-Call Interpretation

Recall we have added another constructor to `Exp`:

```
type Exp = ...
         | Call of string * Exp list
```

The interesting case is the new one for `Call`:

```
let rec eval vtable ftable e =
match e with
  ...
  | Call (fname, args) ->
    let vals = List.map(fun arg->eval vtable ftable arg) args
    match lookup fname ftable with
     | None -> raise (MyError ("Unknown function " + fname))
     | Some fundec -> callFun ftable fundec vals
```

1. evaluate function's actual arguments (by `map`)
2. use the **callFun** function (TBD) to evaluate the function on the resulted values (`vals`).

Evaluating a Function

First, a technical aside: the **callFun** and **eval** functions are going to be *mutually recursive* - they will call each other. In F#, this means we have to connect them in a special way, by defining **callFun** just after **eval** and using **and** instead of another **let rec**.

Evaluating a Function

First, a technical aside: the **callFun** and **eval** functions are going to be *mutually recursive* - they will call each other. In F#, this means we have to connect them in a special way, by defining **callFun** just after **eval** and using **and** instead of another **let rec**.

```
// callFun : FunTab -> FunDec -> Value list -> Value
and callFun ftable fundec act_arg_vals =
  match fundec with
    | FunDec (rettype, fname, form_params, body) ->
        let vtable = bindParams form_params act_arg_vals
        let result = eval vtable ftable body
        match (result, rettype) with
          | (IntVal n, Int) -> IntVal n
          | (BoolVal b, Bool) -> BoolVal b
          | _ -> raise (MyError "Return value mismatch")
```

Evaluating a Function

First, a technical aside: the **callFun** and **eval** functions are going to be *mutually recursive* - they will call each other. In F#, this means we have to connect them in a special way, by defining **callFun** just after **eval** and using **and** instead of another **let rec**.

```
// callFun : FunTab -> FunDec -> Value list -> Value
and callFun ftable fundec act_arg_vals =
  match fundec with
    | FunDec (rettype, fname, form_params, body) ->
        let vtable = bindParams form_params act_arg_vals
        let result = eval vtable ftable body
        match (result, rettype) with
          | (IntVal n, Int) -> IntVal n
          | (BoolVal b, Bool) -> BoolVal b
          | _ -> raise (MyError "Return value mismatch")
```

- Note: function evaluation creates a brand new variable table,
- which binds the formal parameters to the actual-argument values.
- Then the body of the function is evaluated in the new `vtable`.

Binding Function Parameters

```
// bindParams : Param list -> Value list -> VarTab
let rec bindParams params vals =
  match (params, vals) with
    | ([], []) -> empty()  // SymTab.empty()
    | ((par_type, par_name)::rparams, v::rvs) ->
      let vtable = bindParams rparams rvs
      match (par_type, v) with
       | (Int, IntVal  _) -> bind par_name v vtable
       | (Bool,BoolVal _) -> bind par_name v vtable
       | _ -> raise (MyError "Parameter type mismatch")
    | _ -> raise (MyError "Parameter count mismatch")
```

Binding Function Parameters

```
// bindParams : Param list -> Value list -> VarTab
let rec bindParams params vals =
    match (params, vals) with
      | ([], []) -> empty()   // SymTab.empty()
      | ((par_type, par_name)::rparams, v::rvs) ->
          let vtable = bindParams rparams rvs
          match (par_type, v) with
            | (Int,  IntVal  _) -> bind par_name v vtable
            | (Bool,BoolVal _) -> bind par_name v vtable
            | _ -> raise (MyError "Parameter type mismatch")
      | _ -> raise (MyError "Parameter count mismatch")
```

- The `bindParams` function is simple (mostly error detection).
- Recursively binds formal-parameter names to corresponding actual-argument values in a new variable symbol table.
- Error: if the type of a formal parameter is different than the type of the corresponding actual parameter value.
- Error: if the number of formal and actual parameters differs.

Interpreting an Entire Program

How do we start evaluation now? A program is no longer just a single expression, but a list of function declarations.

Interpreting an Entire Program

How do we start evaluation now? A program is no longer just a single expression, but a list of function declarations. Decision: a program is interpreted by calling a function named `main` with zero parameters!

Interpreting an Entire Program

How do we start evaluation now? A program is no longer just a single expression, but a list of function declarations. Decision: a program is interpreted by calling a function named `main` with zero parameters!

First, we define a function for getting a function table from a list of `FunDecs`:

```
// makeFunTable : FunDec list -> FunTab
let rec makeFunTable fundecs =
  match fundecs with
    | [] -> empty ()   // SymTab.empty
    | (fundec::funs) ->
        let ftable = makeFunTable funs
        match fundec with
          | FunDec (rettype, fname, params, body) ->
              bind fname fundec ftable   // SymTab.bind
```

Missing error checking: multiple functions with the same name; (maybe:) multiple parameters of a single function with the same name.

Interpreting an Entire Program

Now we can put all the pieces together into quite a simple `evalProg` function:

```
// evalProg : Prog -> Value
let evalProg prog =
  let ftable = makeFunTable prog
  match lookup "main" ftable with
    | None -> raise (MyError "No main function defined")
    | Some fundec -> callFun ftable fundec []
```

And It Works!

```
let fact_10_prog =
  [FunDec (Int, "main", [],
           Call ("fact", [Constant (IntVal 10)]));
   FunDec (Int, "fact", [(Int, "x")],
           If (Equal (Var "x", Constant (IntVal 0)),
                Constant (IntVal 1),
                Times (Var "x",
                       Call ("fact",
                             [Minus (Var "x",
                                     Constant (IntVal 1))]))))
           ]

- evalProg fact_10_prog
> val it = IntVal 3628800 : Value
```

Notations Used for Interpretation

- Interpreter (and other phases, later in book) expressed in a quasi-functional language.
  - But dynamically typed: think Python or Javascript, not F#.
- Logically split the abstract-syntax representation (AbSyn) into different *syntactic categories*
  - Expressions, function decls, etc.
- Implementing the interpreter $\equiv$ implementing each syntactic category via a function, by case analysis of AbSyn-type constructors.
  - For compactness and readability, represent object-language snippets using concrete syntax.
- For symbols representing names, numbers, etc., use special functions that return these values
  - e.g., *name*(**id**) and *value*(**num**).
- If an error occurs, we call the function **error()** that ends interpretation.
  - In practice, would normally give a meaningful error message here.

Symbol Tables Used by the Interpreter

vtable binds variable names to their AbSyn values. A value is either an integer, character or boolean, or an array literal (of values).
An AbSyn value "knows" its type.

- For atomic values: obvious from shape
- For array values: what if the array is empty?
    - Explicitly keep element-type as part of value

ftable binds function names to their definitions, i.e., the AbSyn representation of a function.

Interpreting Expressions (Part 1)

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
num	$value(\textbf{num})$
id	$v = lookup(vtable, name(\textbf{id}))$ *if* ($v == unbound$) *then* **error**() *else* v
Exp_1 `==` Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$ *if (v_1 and v_2 are values of the same basic type)* *then* ($v_1 == v_2$) *else* **error**()
Exp_1 `+` Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$ *if (v_1 and v_2 are integers) then* ($v_1 + v_2$) *else* **error**()
...	

Interpreting Expressions (Part 2)

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
...	

`if` Exp_1
`then` Exp_2
`else` Exp_3
	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$
	if (v_1 is a boolean value)
	*then if ($v_1 ==$ **true**)*
	    then $Eval_{Exp}(Exp_2, vtable, ftable)$
	    else $Eval_{Exp}(Exp_3, vtable, ftable)$
	else **error**()

`let` **id** `=` Exp_1 `in` Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, name(\mathbf{id}), v_1)$ $Eval_{Exp}(Exp_2, vtable', ftable)$
id ($Exps$)	$def = lookup(ftable, name(\mathbf{id}))$ *if (* def $==$ *unbound) then* **error**() *else* $args = Eval_{Exps}(Exps, vtable, ftable)$ $Call_{Fun}(def, args, ftable)$

Intuitively, $Eval_{Exps}$ evaluates a list of expressions.
$Call_{Fun}$, introduced later, interprets a function call.

Interpreting Expressions (Part 3)

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
`...`	
`iota(` Exp `)`	$len = Eval_{Exp}(Exp, vtable, ftable)$ *if (len is an integer and len* ≥ 0 *)* *then* $[0, 1, .., len - 1]$ *else* **error**()
`reduce(` **id**, Exp_1, Exp_2 `)`	$nel = Eval_{Exp}(Exp_1, vtable, ftable)$ $arr = Eval_{Exp}(Exp_2, vtable, ftable)$ *fdcl = lookup(ftable, name(***id***))* *if (fdcl = unbound) then* **error**() *else if (arr is an array literal)* *then fold (fun a v* $\rightarrow$ $Call_{Fun}$*(fdcl,[a,v],ftable))* *nel arr* *else* **error**()

Interpreting Expressions (Part 3)

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
...	
`iota(` Exp `)`	$len = Eval_{Exp}(Exp, vtable, ftable)$ *if (len is an integer and* $len \geq 0$ *)* *then* $[0, 1, .., len - 1]$ *else* **error**()
`reduce(` **id**, Exp_1, Exp_2 `)`	$nel \;= Eval_{Exp}(Exp_1, vtable, ftable)$ $arr \;= Eval_{Exp}(Exp_2, vtable, ftable)$ *fdcl = lookup(ftable, name(***id***))* *if (fdcl = unbound) then* **error**() *else if (arr is an array literal)* *then fold (fun a v* $\rightarrow$ $Call_{Fun}$*(fdcl,[a,v],ftable))* *nel arr* *else* **error**()

Host-lang facilities: F#'s `fold` used to implement Fasto's `reduce`.

Function-Call Interpretation

$Call_{Fun}(Fun, args, ftable) = $ `case` Fun `of`	
$Type$ **id** ($TypeIds$) $=$ Exp	$vtable = Bind_{TypeIds}(TypeIds, args)$ $v_1 = Eval_{Exp}(Exp, vtable, ftable)$ if (v_1 matches $Type$) then v_1 $else$ **error**()

Function-Call Interpretation

$Call_{Fun}(Fun, args, ftable) =$ `case` Fun `of`	
$Type$ **id** $($ $TypeIds$ $)$ $=$ Exp	$vtable = Bind_{TypeIds}(TypeIds, args)$ $v_1 = Eval_{Exp}(Exp, vtable, ftable)$ if (v_1 matches $Type$) then v_1 $else$ **error**()

- create a *new vtable* by binding the formal to the (already evaluated) actual parameters.
- interpret the expression corresponding to the function's body,
- check that the result value matches the function's return type.

Implementation shortcoming of reduce?

Function-Call Interpretation

$Call_{Fun}(Fun, args, ftable) = \texttt{case}\ Fun\ \texttt{of}$	
$Type\ \mathbf{id}\ (\ TypeIds\ )\ =\ Exp$	$vtable = Bind_{TypeIds}(TypeIds, args)$ $v_1 = Eval_{Exp}(Exp, vtable, ftable)$ $if\ (\ v_1\ \text{matches}\ Type\ )\ \text{then}\ v_1$ $else\ \ \mathbf{error}()$

- create a *new vtable* by binding the formal to the (already evaluated) actual parameters.
- interpret the expression corresponding to the function's body,
- check that the result value matches the function's return type.

Implementation shortcoming of reduce?Its type is not verified!

Initializing vtable: Binding Formal to Actual Params

Error if:

1: two formal parameters have the same name, or if

2: the actual parameter value, v, does not match the declared type of the formal parameter, *Type*.

$Bind_{TypeIds}(TypeIds,\ args) = \texttt{case}\ (TypeIds,\ args)\ \texttt{of}$	
$(\ Type\ \mathbf{id},$ $[v]$ $)$	$if\ (\ v\ \text{matches}\ Type\ )$ $then\ \ bind(empty(), name(\mathbf{id}), v)$ $else\ \ \mathbf{error}()$
$(\ (Type\ \mathbf{id},$ $TypeIds),$ $v :: vs$ $)$	$vtable = Bind_{TypeIds}(TypeIds,\ vs)$ $if\ \ lookup(vtable, name(\mathbf{id})) = unbound$ $and\ \ v\ \text{matches}\ Type$ $then\ \ bind(vtable, name(\mathbf{id}), v)$ $else\ \ \mathbf{error}()$
$\_\_$	$\mathbf{error}()$

Interpreting the Whole Program

$Run_{Program}(Program, input) =$ `case` $Program$ `of`	
$Funs$	$ftable = Build_{ftable}(Funs)$ $def = lookup(ftable,$ `"main"`$)$ $if\ (\ def == unbound\ )\ \ then$ **error**() $else\ \ Call_{Fun}(def, input, ftable)$

$Build_{ftable}(Funs) =$ `case` $Funs$ `of`	
Fun	$f = Get_{fname}(Fun)$ $bind(empty(), f, Fun)$
$Fun\ Funs$	$ftable = Build_{ftable}(Funs)$ $f = Get_{fname}(Fun)$ $if\ (\ lookup(ftable, f) == unbound\ )$ $then\ \ bind(\ ftable, f, Fun\ )$ $else$ **error**()

$Get_{fname}(Fun) =$ `case` Fun `of`	
Type **id** (*TypeIds*) = *Exp*	name(**id**)

What's next

- LAB today: walk-through of Fasto's AbSyn and Interpreter
  - Possibly with another quick look at other parts, as well
- Tomorrow: **DIKU Bits** talk, Lille UP1, 12:15–13:00
  - Troels Henriksen: "Deception is OK when you gotta go fast"
  - Highly relevant as background/inspiration for IPS!
- Wednesday lecture: type checking and type inference
  - With Robert!
- Wednesday exercise classess
  - Chance to get last-minute in-person help with Weekly 1
  - Have a look at the suggested exercises beforehand
- Next Monday: intermediate-code generation