Interpretation

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2012 Compiler Lecture Notes

Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree → Intepretation

↓

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

Fasto Language: Function Declaration and Types

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Funs} \\[1em]
\mathit{Funs} & \rightarrow & \mathit{Fun} \\
\mathit{Funs} & \rightarrow & \mathit{Fun}\ \mathit{Funs} \\[1em]
\mathit{Fun} & \rightarrow & \mathit{Type}\,\mathbf{id}\ (\ \mathit{TypeIds}\ )\ =\ \mathit{Exp} \\[1em]
\mathit{TypeIds} & \rightarrow & \mathit{Type}\ \mathbf{id} \\
\mathit{TypeIds} & \rightarrow & \mathit{Type}\ \mathbf{id}\ ,\ \mathit{TypeIds} \\[1em]
\mathit{Type} & \rightarrow & \texttt{int} \\
\mathit{Type} & \rightarrow & \texttt{char} \\
\mathit{Type} & \rightarrow & \texttt{bool} \\
\mathit{Type} & \rightarrow & [\mathit{Type}] \\[1em]
\mathit{Exps} & \rightarrow & \mathit{Exp} \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\ ,\ \mathit{Exps}
\end{array}
$$

- First-order functional language & mutually recursive functions.
- Program starts by executing "main", which takes no args.
- Separate namespaces for vars & funs.
- Illegal for two formal params of the same function to share the same name.
- Illegal for two functions to share the same name.

Fasto Language: Basic Expressions

$$
\begin{array}{lll}
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{charlit} \\
 & & \\
Exp & \rightarrow & Exp + Exp \\
Exp & \rightarrow & Exp - Exp \\
Exp & \rightarrow & Exp < Exp \\
Exp & \rightarrow & Exp = Exp \\
 & & \\
Exp & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Exp\ \texttt{else}\ Exp \\
Exp & \rightarrow & \texttt{let}\ \textbf{id} = Exp\ \texttt{in}\ Exp \\
Exp & \rightarrow & \textbf{id}\ () \\
Exp & \rightarrow & \textbf{id}\ (\ Exps\ )
\end{array}
$$

- +, - defined on ints.
- =, < defined on basic-type values.
- Static Scoping: `let` bindings and function declarations create new scopes.
- A `let id ...` may hide an outer-scope var also named `id`.
- Call by Value.

Demonstrating Recursive Calls and IO in Fasto

Fibonacci Example and Use of Read/Write

```
fun int fibo(int n) = if         (n = 0) then 1
                      else if (n = 1) then 1
                      else fibo(n-1) + fibo(n-2)

fun int main () = let w = write("Enter Fibonacci's number: \ n")
                  in  let n = read(int)
                      in let w = write("Result is: \ n")
                         in let ww = write(w)  //what is printed?
                            in write( fibo(n) )
```

Demonstrating Recursive Calls and IO in Fasto

Fibonacci Example and Use of Read/Write

```
fun int fibo(int n) = if          (n = 0) then 1
                      else if (n = 1) then 1
                      else fibo(n-1) + fibo(n-2)

fun int main () = let w = write("Enter Fibonacci's number: \ n")
                  in  let n = read(int)
                      in let w = write("Result is: \ n")
                         in let ww = write(w)  //what is printed?
                            in write( fibo(n) )
```

Polymorphic Functions `read` and `write`:

- the only constructs in FASTO exhibiting side-effects (IO).
- valid uses of read: `read(int)`, `read(char)`, or `read(bool)`; takes a type parameter and returns a (read-in) value of that type.
- `write` : $\alpha \rightarrow \alpha$, where α can be `int`, `char`, `bool`, `[char]`, or **stringlit**. *`write` returns (a copy of) its input parameter.*

Fasto Language: Array Constructors & Combinators

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \texttt{read}\ (\ \mathit{Type}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{write}\ (\ \mathit{Exp}\ ) \\
\mathit{Exp} & \rightarrow & \textbf{stringlit} \\
\mathit{Exp} & \rightarrow & \{\ \mathit{Exps}\ \} \\
\mathit{Exp} & \rightarrow & \texttt{iota}\ (\ \mathit{Exp}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{replicate}\ (\ \mathit{Exp}\ ,\ \mathit{Exp}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{map}\ (\ \textbf{id}\ ,\ \mathit{Exp}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{reduce}\ (\ \textbf{id}\ ,\ \mathit{Exp}\ ,\ \mathit{Exp}\ ) \\
\mathit{Exp} & \rightarrow & \textbf{id}\ [\ \mathit{Exp}\ ]
\end{array}
$$

- `read` / `write` polymorphic operators,
- array constructors: string and array literals, `iota`, `replicate`,
- second-order array combinators (SOAC): `map` and `reduce`,
- array indexing: check if index is within bounds.

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello"` ≡ `{'H','e','l','l','o'} : [[int]]`

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello"` $\equiv$ `{'H','e','l','l','o'} : [[int]]`
- `iota(n)` $\equiv$ `{0,1,2,...,n-1}`; *type of* `iota : int` $\rightarrow$ `[int]`

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello"` $\equiv$ `{'H','e','l','l','o'} : [[int]]`
- `iota(n)` $\equiv$ `{0,1,2,...,n-1}`; *type of* `iota : int` $\rightarrow$ `[int]`
- `replicate(n, q)` $\equiv$ `{q, q, ..., q}`, i.e., an array of size `n`,
  type of `replicate : int *` α $\rightarrow$ `[`α`]`.
  What is the result of `replicate(2, {8, 9})` ?

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello" ≡ {'H','e','l','l','o'} : [[int]]`
- `iota(n) ≡ {0,1,2,...,n-1}`; *type of* `iota : int` $\rightarrow$ `[int]`
- `replicate(n, q) ≡ {q, q, ..., q}`, i.e., an array of size `n`,
  type of `replicate : int` $* \ \alpha \rightarrow [\alpha]$.
  What is the result of `replicate(2, {8, 9})` ?

Second-Order Array Combinators:

- `map(f,` $\{x_1, \ldots, x_n\}$`) = {f(`x_1`, .., f(`x_n`))}`, where type of $x_i : \alpha$, of `f` $: \alpha \rightarrow \beta$, of `map` $: (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello"` $\equiv$ `{'H','e','l','l','o'} : [[int]]`
- `iota(n)` $\equiv$ `{0,1,2,...,n-1}`; *type of* `iota : int` $\rightarrow$ `[int]`
- `replicate(n, q)` $\equiv$ `{q, q, ..., q}`, i.e., an array of size `n`, *type of* `replicate : int *` $\alpha \rightarrow [\alpha]$.
  What is the result of `replicate(2, {8, 9})` ?

Second-Order Array Combinators:

- `map(f,` $\{x_1, .., x_n\}$`)` = $\{f(x_1, .., f(x_n))\}$, where type of $x_i : \alpha$, of $f : \alpha \rightarrow \beta$, of `map` $: (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$
- `reduce(`$\odot$`, e,` $\{x_1, x_2, .., x_n\}$`)` = $(..(e \odot x_1)..\odot x_n)$, where type of $x_i : \alpha$, of $e : \alpha$, type of $\odot : \alpha * \alpha \rightarrow \alpha$
  type of `reduce` $: (\alpha * \alpha \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$.

Demonstrating Map-Reduce Programming

Can We Write Main Using Map and Reduce?

foldl : $(\alpha * \beta \rightarrow \beta) * \beta * [\alpha] \rightarrow \beta$

foldl($\odot$, e, $\{x_1, .., x_n\}$) $\equiv$ ($x_n \odot$ .. ($x_1 \odot$ e) ..)

```
fun bool f(int a, bool b) = (a > 0) && b
fun bool main() = let x = {1, 2, 3}
                  in foldl(f, True, x)
```

Demonstrating Map-Reduce Programming

Why Doe Fasto *Not* Support Foldl?

foldl : $(\alpha * \beta \rightarrow \beta) * \beta * [\alpha] \rightarrow \beta$

foldl($\odot$, e, {x_1, .., x_n}) $\equiv$ (x_n $\odot$ .. (x_1 $\odot$ e) ..)

```
fun bool f(int a, bool b) = (a > 0) && b
fun bool main() = let x = {1, 2, 3}
                  in foldl(f, True, x)
```

Because It Is Typically The Composition of a Map With a Reduce:

map : $(\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$

map (f, {x_1, .., x_n}) $\equiv$ {f(x_1), .., f(x_1))}

reduce : $(\alpha * \alpha \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$

reduce ($\odot$, e, {x_1, .., x_n}) $\equiv$ (x_n $\odot$ .. (x_1 $\odot$ e) ..)

```
fun bool f(int a) = a > 0
fun bool main() = let x = {1, 2, 3} in
                  let y = map(f, x)
                  in reduce(op &&, True, y)
```

Array Combinators & Read and Write

What Is Printed If The User Types in: “1 2 8 9”?

```
fun  int writeInt   ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, replicate(2,2)) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

Array Combinators & Read and Write

What Is Printed If The User Types in: "1 2 8 9"?

```
fun  int writeInt   ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, replicate(2,2)) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

`readIntArr(n)` "reads" a $1D$ array of n elements.

`map(readIntArr,replicate(2,2))` $\equiv$ `map(readIntArr,2,2)` $\equiv$ `{readIntArr(2),readIntArr(2)}` builds $2D$ array `{{1,2},{8,9}}`.

Array Combinators & Read and Write

What Is Printed If The User Types in: "1 2 8 9"?

```
fun  int writeInt   ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, replicate(2,2)) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

So, `map(readIntArr,replicate(2,2))` ≡ `{{1,2},{8,9}}`.

`reduce( plusV2, {0,0}, {{1,2},{8,9}} )` ≡

Array Combinators & Read and Write

What Is Printed If The User Types in: “1 2 8 9”?

```
fun  int writeInt   ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, replicate(2,2)) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

So, `map(readIntArr,replicate(2,2))` $\equiv$ {{1,2},{8,9}}.

`reduce( plusV2, {0,0}, {{1,2},{8,9}} )` $\equiv$
`plusV2( plusV2({0,0}, {1,2}), {8,9} )` $\equiv$
`plusV2( {0+1,0+2}, {8,9} )` $\equiv$ {1+8, 2+9} $\equiv$ {9,11}.

Finally, `map(writeInt,arr1)` prints the elements of `arr1`, i.e., 9 11!

Interpretation Intuition: Solving First-Grade Math

Term Rewriting *vs.* Interpretation

```
q = (7 + 5) / 3 + (7 + 8) / 5
  = 12 / 3      + (7 + 8) / 5
  = 4           + (7 + 8) / 5
  = 4           + 15 / 5
  = 4           + 3

  = 7
```

```
x  = 7 + 5      ( = 12 )
y1 = x / 3      ( = 12 / 3 = 4)

x  = 7 + 8      ( = 12)
y2 = x / 5      ( = 15 / 5 = 3)

q  = y1 + y2    ( = 4 + 3 = 7 )
```

Interpretation Intuition: Solving First-Grade Math

Term Rewriting *vs.* Interpretation

```
q = (7 + 5) / 3 + (7 + 8) / 5
  = 12 / 3      + (7 + 8) / 5
  = 4           + (7 + 8) / 5
  = 4           + 15 / 5
  = 4           + 3

  = 7
```

$$
\begin{array}{lll}
x & = 7 + 5 & (= 12) \\
y_1 & = x / 3 & (= 12 / 3 = 4) \\
 & & \\
x & = 7 + 8 & (= 12) \\
y_2 & = x / 5 & (= 15 / 5 = 3) \\
 & & \\
q & = y_1 + y_2 & (= 4 + 3 = 7)
\end{array}
$$

Let (Partial) Semantics

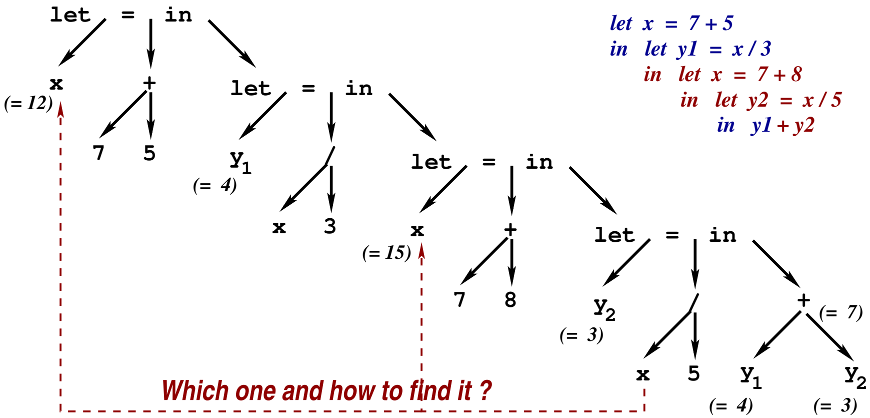
let t = e_1 in e_2

Semantics: *evaluate* e_1, *replace* t *with* e_1*'s value in* e_2, *and evaluate* e_2

Program to evaluate q

```
let x = 7 + 5
in  let y1 = x / 3
    in  let x = 7 + 8
        in  let y2 = x / 5
            in  y1 + y2
```

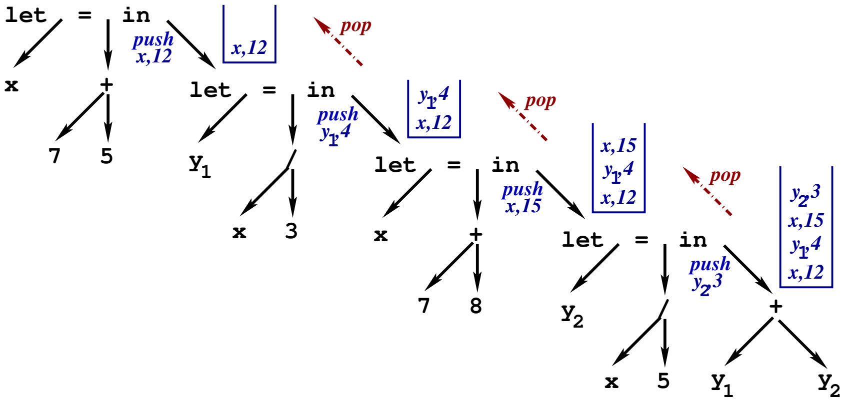
How to Interpret? Lang. Semantics + Symbol Table



Semantics: The use of `x` refers to which of the two variables named `x`?

Symbol Table: How to keep track of the values of various variables?

How to Interpret? Lang. Semantics + Symbol Table



Semantics: The use of x refers to the "closest"-outer scope that provides a definition for x.

Symbol Table: the implementation uses a stack, which is scanned top down and returns the first encountered binding of x.

Symbol Table

Symbol Table: binds names to associated information.

Operations:

- *empty:* empty table, i.e., no name is defined.
- *bind:* records a new `(name,info)` association. If `name` already in the table, the new binding takes precedence.
- *look-up:* finds the information associated to a name. The result must indicate also whether the name was present in the table.
- *enter*s a new scope: semantically adds new bindings.
- *exit*s a scope: restores the table to what it has been before entering the current scope.

For Interpretation: what is the info associated with a named variable?

Symbol Table

Easiest implementation is a stack: (i) *bind* pushes a new binding, (ii) *lookup* searches the stack top down, (iii) *enter* pushes a marker, (iv) *exit* pops all elements up-to-and-including the first marker. Example!

Implementation in a Functional Language

```
fun empty() = []

fun bind n i stab = (n,i)::stab
```

```
fun lookup n []          = NONE
  | lookup n ((n1,i1)::tab) =
      if   (n=n1) then SOME i1
      else lookup n tab
```

Functional Implementation uses a list:

- *enter* saves the reference of the current (old) table, and creates a new table by appending new bindings to the current symbol table.
- *exit* discards the current table and uses the old table (previously saved in enter).

What is Interpretation?

Compiler	*vs.*	Interpreter
source program ↓ Compiler ↓ *target program*	*input* ↓ Target Program ↓ *output*	*source program* *input* ↓ ↓ Interpreter ↓ *output*

The interpreter directly *executes* one by one the operations specified in the *source program* on the *input* supplied by the user, by using the facilities of its implementation language.

Why interpret? Debugging, Prototype-Language Implementation, etc.

Notations Used for Interpretation

We logically split the abstract-syntax representation (AbSyn) into different *syntactic categories*: expressions, function decl, etc.

Implementing the interpreter $\equiv$ implementing each syntactic category via a function, that uses case analysis of AbSyn-type constructors.

In practice we work on AbSyn, but here we represent interpretation generically by using a notation that resembles the language grammar.

For symbols representing names, numbers, and the like, we 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.

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. An AbSyn value "knows" its type.

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

Interpreting Fasto Expressions (Part 1)

For simplicity, we assume the result is an SML value, e.g., `int`, `bool`.

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

Interpreting Fasto Expressions (Part 2)

$Eval_{Exp}(Exp, vtable, ftable) = \texttt{case}\ Exp\ \texttt{of}$	
...	
`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 Fasto Expressions (Part 3)

For simplicity, we assume the result is an SML value,
e.g., an array literal is represented as a SML list.

$Eval_{Exp}(Exp, vtable, ftable) =$ `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**()
`map(` **id**, Exp `)`	$arr = Eval_{Exp}(Exp, vtable, ftable)$ *fdcl = lookup(ftable, name(***id***))* *if (fdcl = unbound) then* **error**() *else if (arr is an array literal)* *then map (fn x* $\Rightarrow$ $Call_{Fun}$*(fdcl,[x],ftable)) arr* *else* **error***()*

Function-Call Interpretation

- 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.

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

Initializing vtable: Binding Formal to Actual Params

Error iff:

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

2: the actual parameter value, `aarg_val`, does not matches the declared type of the formal parameter, *Type*.

$Bind_{TypeIds}(TypeIds,\ args)$ = `case` $(TypeIds,\ args)$ `of`	
(*Type* `farg_name`, [`aarg_val`])	*if* (`aarg_val matches` *Type*) *then* $bind(empty(),$ `farg_name`, `aarg_val`$)$ *else* **error**()
(*Type* `farg_name`, *TypeIds*, (`aarg_val` :: vs))	$vtable = Bind_{TypeIds}(TypeIds,\ vs)$ *if* $lookup(vtable,$ `farg_name`$) = unbound$ *and* `aarg_val matches` *Type* *then* $bind(vtable,$ `farg_name`, `aarg_val`$)$ *else* **error**()
__	**error**()

Interpreting the Whole Program

$Run_{Program}(Program, input) = \texttt{case}\ Program\ \texttt{of}$	
$Funs$	$ftable\ = Build_{ftable}(Funs)$ $def\ = lookup(ftable, \texttt{"main"})$ $if\ (\ \mathrm{def} = unbound\ )\ \ then\ \ \mathbf{error}()$ $else\ \ Call_{Funs}(def, [input], ftable)$

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

$Get_{fname}(Fun) = \texttt{case}\ Fun\ \texttt{of}$	
$Type\ \texttt{fid}\ (\ TypeIds\ )\ =\ Exp$	$\texttt{fid}$

Interpretation vs. Compilation: Pros and Cons

Think about it for the next lecture!

Parameter Passing Mechanisms

Actual Parameters: the ones used in the function call.

Formal Parameters: the ones used in the function declaration.

Call by Value: The actual parameter is evaluated if it is an expression. The callee logically operates on a copy of the actual parameter, hence any modifications to the formals do not affect the actual parameters.

Call by Reference: the callee operates directly on the actual parameters (callee updates are visible in the caller).

Call by Value Result: Actual parameters passed in as with call by value. Just before the callee returns, all actual parameters are updated to the value of the formal parameters in the callee. Why Useful?

Call by Name: callee executes as if the actual parameters were substituted directly in the code of the callee. Consequences?

Parameter Passing Example

Results under call by (i) value, (ii) reference and (iii) value result?

```
int x = 5;
f(x, x);                    void f(int a, int b)     a = 2*a + b;

print(x);
```

Differ when the actual parameters are variables (as opposed to expressions).

Call by Value: 5. Call by Reference: 0.

Call by Value Result: depending on the update order either 15 or 10.

Parameter Passing Example

Results under call by (i) value, (ii) reference and (iii) value result?

```
int x = 5;                      void f(int a, int b) {
f(x, x);                            a = 2*a + b;
                                    b =   a - b;
print(x);                       }
```

Differ when the actual parameters are variables.

Call by Value: 5. Call by Reference: 0.

Call by Value Result: depending on the update order either 15 or 10.

What is printed under call by (i) Name and (ii) all the rest?

```
void f(int a, int b) {                       //infinite recursion
  if(a > 0) print(a); else print(b);         int g(int a) {
}                                              return g(a);
f(1, g(1));                                  }
```

Differ when the actual parameters are expressions. Call by Name: 1.

All the rest: nothing gets printed, non-terminating program.

Scopes: Statical (Lexical) Scoping

A scope is the context in a program within which a named variable can be used, i.e., a name is *bound* to the variable's storage.

Outside the variable's scope, the name does not refer to that variable, albeit its value may exist in memory (and may even be accessible).

Scopes: Statical (Lexical) Scoping

A scope is the context in a program within which a named variable can be used, i.e., a name is *bound* to the variable's storage.

Outside the variable's scope, the name does not refer to that variable, albeit its value may exist in memory (and may even be accessible).

Static scoping defines a set of rules that allows one (the compiler) to know to what variable a name refers to, just by looking at (a small part of) the program, i.e., without running it. In C the scope of:

- a global variable can be as large as the entire program.
- a local variable can be as large as its declaration block.
- function arguments can be as large as the body of the function.

Is it correct to say "the scope is exactly ..."?

Scopes: Statical (Lexical) Scoping

A scope is the context in a program within which a named variable can be used, i.e., a name is *bound* to the variable's storage.

Outside the variable's scope, the name does not refer to that variable, albeit its value may exist in memory (and may even be accessible).

Static scoping defines a set of rules that allows one (the compiler) to know to what variable a name refers to, just by looking at (a small part of) the program, i.e., without running it. In C the scope of:

- a global variable can be as large as the entire program.
- a local variable can be as large as its declaration block.
- function arguments can be as large as the body of the function.

Is it correct to say "the scope is exactly ..."?

No, because variable names may be reused!

Statical (Lexical) Scoping Example

What is printed?

```
public static void main(String[] args) {                        //scope B1
    int a = 0;
    int b = 0;
    {                                                       //scope B2
        int b = 1;
        {                                           //scope B3
            int a = 1; System.out.println(a+" "+b);
        }                                           //end B3

        {                                           //scope B4
            int b = 2; System.out.println(a+" "+b);
        }                                           //end B4
        System.out.println(a+" "+b);
    }                                                       //end B2
    System.out.println(a+" "+b)
}                                                               //end B1
```

Statical (Lexical) Scoping Example

What is printed?

```
public static void main(String[] args) {                    //scope B1
    int a = 0;     // Scope: B1 - B3
    int b = 0;     // Scope: B1 - B2
    {                                                     //scope B2
        int b = 1; // Scope: B2 - B4
        {                                          //scope B3
            int a = 1; System.out.println(a+" "+b);
        }                                          //end B3

        {                                          //scope B4
            int b = 2; System.out.println(a+" "+b);
        }                                          //end B4
        System.out.println(a+" "+b);
    }                                                     //end B2
    System.out.println(a+" "+b)
}                                                           //end B1
```

Dynamic Scoping

In general: a policy is dynamic if it is based on factors that cannot be known statically, i.e., at compile time. Example: virtual calls in Java.

Dynamic scoping usually corresponds to the following policy: a use of a name x refers to the declaration of x in the most-recently-called and not-yet-terminated function that has a declaration of x.

E.g., early variants of Lisp, Perl.

Dynamic vs. Static Scoping Example

What is printed under static & dynamic scoping, respectively?

```
int x = 1;

void g() { print(x); }

int main() { int x = 2; f(); }
```

```
void f() {
    int y = readint();
    if(y > 5) { int x = 3; g(); }
    else      { g();
}
```

Dynamic vs. Static Scoping Example

What is printed under static & dynamic scoping, respectively?

```
int x = 1;

void g() { print(x); }

int main() { int x = 2; f(); }
```

```
void f() {
    int y = readint();
    if(y > 5) { int x = 3; g(); }
    else      { g();
}
```

Static Scoping: 1, i.e., the global x.

Dynamic Scoping: 3, if y > 5, and 2 otherwise.

Interpreting a Fasto Program – Implementation

```
Fasto.Binding  ≡  (string * Type) (*formal arg name & type*)
Fasto.FunDec  ≡ (string * Type * Binding list * Exp *pos)
```

Entry Point for Interpretation & Building Function's Symbol Table

```
fun getFunRTP (_,rtp,_,_,_)= rtp
fun getFunArgs(_,_,arg,_,_)= arg

(*Fasto.FunDec listFasto.Exp*)
fun eval_pgm funlst =
  let val ftab= buildFtab funlst
      val main= SymTab.lookup
                  "main" ftab
  in case main of
       NONE => raise Error(
         "Undefined Main!",(0,0))
     | SOME f =>
         call_fun(f,[],ftab,
                    getFunPos(f) )
  end
```

```
fun getFunName(fid,_,_,_,_)= fid
fun getFunPos (_,_,_,_,pos)= pos
fun getFunBody(_,_,_,bdy,_)= bdy

fun buildFtab [] = empty()
| buildFtab ( fdcl::fs ) =
  let val ftab= buildFtab fs
      val fid = getFunName(fdcl)
      val pos = getFunPos (fdcl)
      val f   = lookup fid ftab
  in case f of
       NONE => bind fid fdcl ftab
     | SOME fdecl => raise Error
        ("Duplicate Fun: "^fid,pos)
  end
```

Interpreting a Function Call – Implementation

Interpreting a Function Call & Helper Function "bindTypeIds"

```
fun call_fun(
  (fid,rtp,farg,body,pdcl)
  aarg, ftab, pcall     ) =

  (* build args' SymTab *)
let val vtab=bindTypeIds (
                farg, aarg
              )
     (* eval fun's body *)
    val res = eval_exp (
            body,vtab,ftab
          )
 (* check result  value *)
 (* matches return type *)
in if(typeMatch(rtp,res))
   then res
   else raise Error(
   "Illegal Result Type!")
end
```

```
fun bindTypeIds([], []) =
      SymTab.empty()
  | bindTypeids([], aa) =
      raise Error("Illegal Args Num!")
  | bindTypeIds(bb, []) =
      raise Error("Illegal Args Num!")
  | bindTypeIds( (faid, fatp)::farg,
                             aa::aarg )
= let val vtab = bindTypeIds(farg,aarg)
      val arg = SymTab.lookup faid vtab
  in if( typeMatch(fatp, aa) )
     then case arg of
            NONE =>
              SymTab.bind faid aa vtab
          | SOME m => raise Error (
              "Duplicate Formal Arg" )
     else raise Error(
           "Illegal Arg Value/Type!" )
  end
```

Checking Whether a Value Matches Its Type

- Value of type `int` is `Fasto.Num(i, pos)`,
- Value of type `bool` is `Fasto.Log(b, pos)`,
- Value of type `char` is `Fasto.CharLit(ch, pos)`.

Recursive Check for Array Values

```
fun typeMatch ( Int  (p), Num    (i,  p2) ) = true
  | typeMatch ( Bool (p), Log    (b,  p2) ) = true
  | typeMatch ( Char (p), CharLit(ch, p2) ) = true

  | typeMatch ( Array(t,p), ArrayLit(arr, tp, p2) ) =
      (** ...... fill in the blanks ......... **)
      (** ...... fill in the blanks ......... **)

  | typeMatch (_, _) = false
```

Checking Whether a Value Matches Its Type

Recursive Check for Array Values

```
fun typeMatch ( Int  (p), Num    (i,  p2) ) = true
  | typeMatch ( Bool (p), Log    (b,  p2) ) = true
  | typeMatch ( Char (p), CharLit(ch, p2) ) = true

  | typeMatch ( Array(t,p), ArrayLit(arr, tp, p2) ) =
      let val mlst = map (fn x => typeMatch(t,x)) arr
      in  foldl (fn(x,y) => x andalso y) true mlst

  | typeMatch (_, _) = false
```

Interpreting a Fasto Expression – Implementation

The result of interpreting an expression is a value-AbSyn node:

- `Fasto.Num(i, pos)` represents integer `i`,
- `Fasto.Log(b, pos)` represents boolean `b`,
- `Fasto.CharLit(ch, pos)` represents character `ch`.

Interpreting a Value, Addition, Let Construct & Helper Function

```
fun eval_exp(Num(n,pos), vtab, ftab)=
      Num(n1,p1)
  | eval_exp(Plus(e1,e2,p), vtab, ftab)=
      let val r1 = eval_exp(e1,vtab,ftab)
          val r2 = eval_exp(e2,vtab,ftab)
      in evalBinop(op +, r1, r2, p)
      end
  | eval_exp( Let( Dec(id,e,p), exp,pos )
              , vtab, ftab ) =
      let val r = eval_exp(e, vtab, ftab)
          val nvtab = SymTab.bind id r vtab
      in eval_exp(exp, nvtab, ftab)
      end
```

```
fun evalBinop( bop
               Num(n1,p1),
               Num(n2,p2),
               pos
             )         =
    Num(bop(n1, n2), pos)

  | evalBinop(_,_,_,_) =
    raise Error
        ("Illegal binop")
```

Interpreting a Fasto Expression – Implementation

Array literals: `data Exp = ArrayLit of Exp list * Type * pos`

Interpreting a Function Call and a Reduction

```
| eval_exp ( ArrayLit (l, t, pos), vtab, ftab ) =
      let val els = (map (fn x => eval_exp(x, vtab, ftab)) l)
      in ArrayLit(els, t, pos)
      end
...
| eval_exp( Apply(fid, aas, p), vtab, ftab ) =
      let val evargs = map (fn e => eval_exp(e,vtab,ftab)) aas
      in case(SymTab.lookup fid ftab) of
           SOME f => call_fun(f,evargs,ftab,p)
         | NONE   => raise Error("SymTabErr!")
      end
```

BUG 1?

Interpreting a Fasto Expression – Implementation

Array literals: `data Exp = ArrayLit of Exp list * Type * pos`

Interpreting a Function Call and a Reduction

```
| eval_exp ( ArrayLit (l, t, pos), vtab, ftab ) =
      let val els = (map (fn x => eval_exp(x, vtab, ftab)) l)
      in ArrayLit(els, t, pos)
      end
...
| eval_exp( Apply(fid, aas, p), vtab, ftab ) =
      let val evargs = map (fn e => eval_exp(e,vtab,ftab)) aas
      in case(SymTab.lookup fid ftab) of
             SOME f => call_fun(f,evargs,ftab,p)
           | NONE   => raise Error("SymTabErr!")
      end
```

BUG 1? should compute the element type `t`. BUG 2? should check that all elements have the same type `t`.

Interpreting Fasto-Reduce Expression – Implem.

Array literals: `data Exp = ArrayLit of Exp list * Type * pos`

Interpreting a Function Call and a Reduction

```
(* REMEMBER: reduce( ⊙, e, { x1,..,xn }) ≡ (..(e ⊙ x1 ) .. ⊙ xn ) *)
  | eval_exp ( Reduce(fid, ne, alit, t, p), vtab, ftab ) =
      let val fdcl = SymTab.lookup fid ftab
          val arr  = eval_exp(alit, vtab, ftab)
          val nel  = eval_exp(ne,   vtab, ftab)
          val f    = case fdcl of
                        SOME m => m
                      | NONE   => Error("SymTabErr")
      in case arr of
           ArrayLit(lst,ti,pi) =>
               foldl ( fn(x,y) => call_fun(f,[x,y],ftab,p) ) nel lst
         | otherwise => error("Illegal Value")
```

BUG? not checking that the type of `nel` ≡ the element type of `lst`!

Interpreting Fasto-Reduce Expression – Implem.

Our simple interpretation detects type mismatches (only) when functions/operators are called!

Bugs?

```
fun bool f(int i, bool b) = if(b) then (i = 0) else (i = 1)

fun bool main() =
  let arr = {1, 2, 3} in
    reduce(f, (1=1), arr) // BUG

fun [int] main() =
  let z = { 1, 2, chr(3) } in //type error not detected
     { 1, 2, 3 }

fun [int] main() =
  let z = { 1, 2, chr(3) } in
      z //type error detected when we use/return z
```