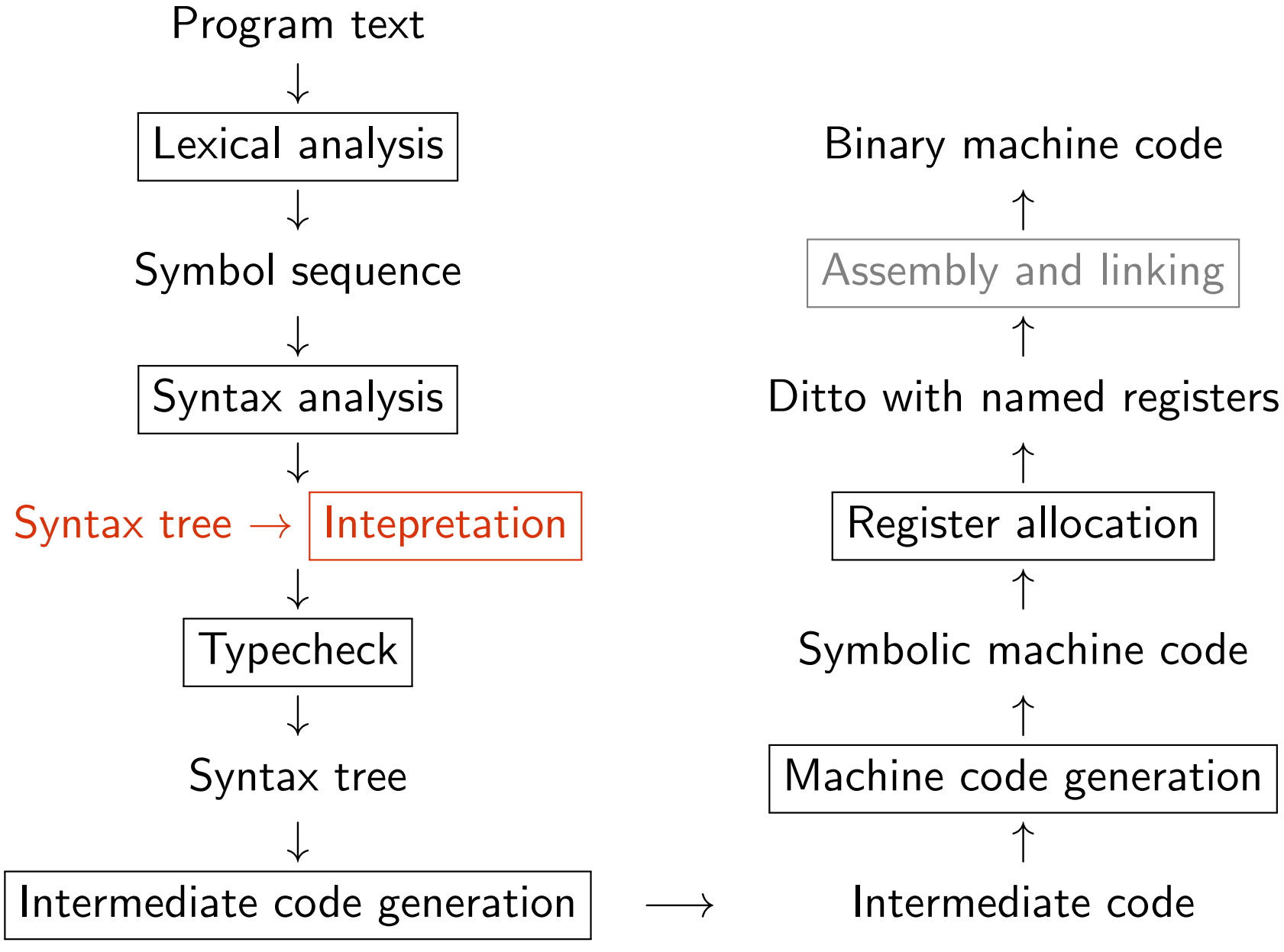
Interpretation

Cosmin E. Oancea
cosmin@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2013 Compiler Lecture Notes

Structure of a Compiler



1. A Simple Functional Language & Its Semantics
2. Interpretation: Intuition, Symbol Tables & Strategy
3. Generic Interpretation (Using Book Notations)
4. Parameter Passing, Static *vs.* Dynamic Scoping
5. Interpretation in the Context of Paladim Language

Function Declaration and Types

Program → *Funs*

Funs → *Fun*
Funs → *Fun Funs*

Fun → *Type* **id** (*TypeIds*) = *Exp*

TypeIds → *Type* **id**
TypeIds → *Type* **id** , *TypeIds*

Type → `int`
Type → `char`
Type → `bool`

Exps → *Exp*
Exps → *Exp* , *Exps*

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

Expressions

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \textbf{id} \\
\mathit{Exp} & \rightarrow & \textbf{num} \\
\mathit{Exp} & \rightarrow & \textbf{charlit}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \mathit{Exp} + \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} - \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} < \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} = \mathit{Exp}
\end{array}
$$

$$
\mathit{Exp} \rightarrow \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp}
$$

$$
\mathit{Exp} \rightarrow \texttt{let}\ \textbf{id} = \mathit{Exp}\ \texttt{in}\ \mathit{Exp}
$$

$$
\mathit{Exp} \rightarrow \textbf{id}\ (\ \mathit{Exps}\ )
$$

- +, - 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.

Example of a Program in Our Simple Language

What Is This Function Computing ?

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

int main () =
    let n = 8
    in  what_is_this_fun(n)
```

Example of a Program in Our Simple Language

What Is This Function Computing ?

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

int main () =
    let n = 8
    in  what_is_this_fun(n)
```

$$x_0 = 0,\ x_1 = 1,\ \ldots,\ x_n = x_{n-1} + x_{n-2}$$

$$x_2 = x_1 + x_0 = 1,\ x_3 = x_2 + x_1 = 2,\ x_4 = x_3 + x_2 = 3,\ x_5 = 5, \ldots$$

Example of a Program in Our Simple Language

What Is This Function Computing ?

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

int main () =
    let n = 8
    in  what_is_this_fun(n)
```

$$x_0 = 0,\ x_1 = 1,\ \ldots,\ x_n = x_{n-1} + x_{n-2}$$

$$x_2 = x_1 + x_0 = 1,\ x_3 = x_2 + x_1 = 2,\ x_4 = x_3 + x_2 = 3,\ x_5 = 5, \ldots$$

Fibonacci

Interpretation Intuition: Solving First-Grade Math

Term Rewriting *vs.* Interpretation

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

  = 7
```

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

Interpretation Intuition: Solving First-Grade Math

Term Rewriting *vs.* Interpretation

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

Let (Partial) Semantics

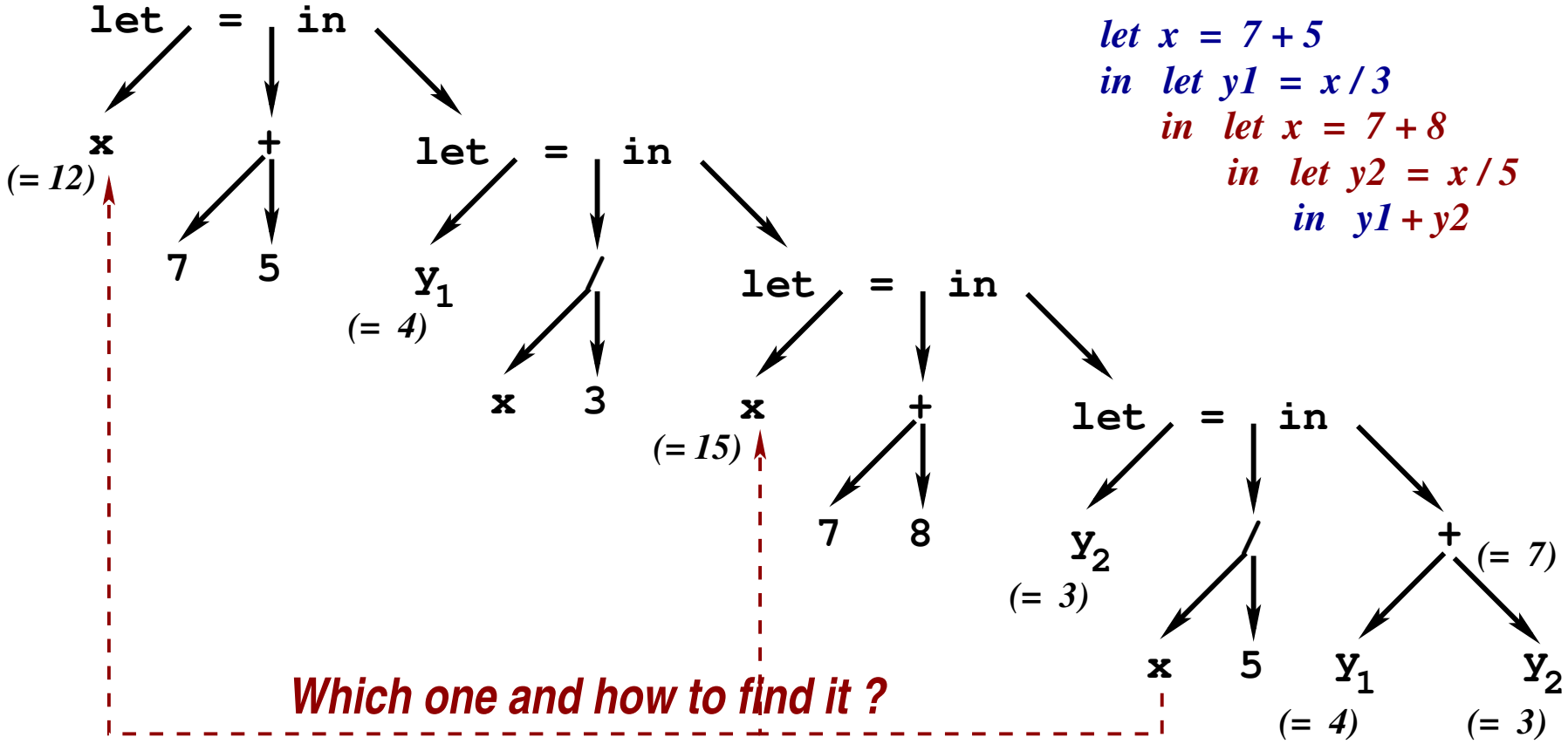
`let t = e1 in e2`

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
```

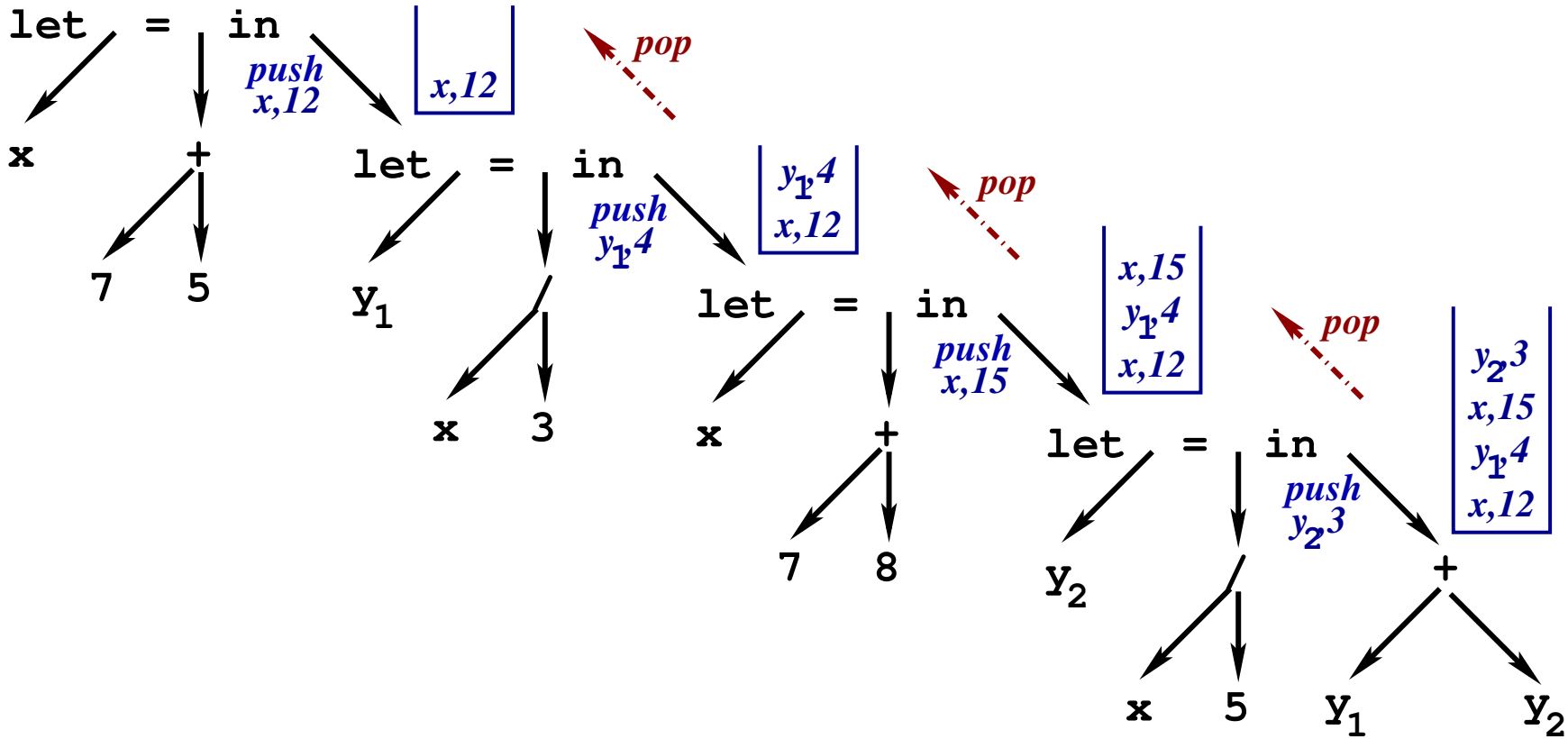
How to Interpret? Lang. Semantics + Symbol Table



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

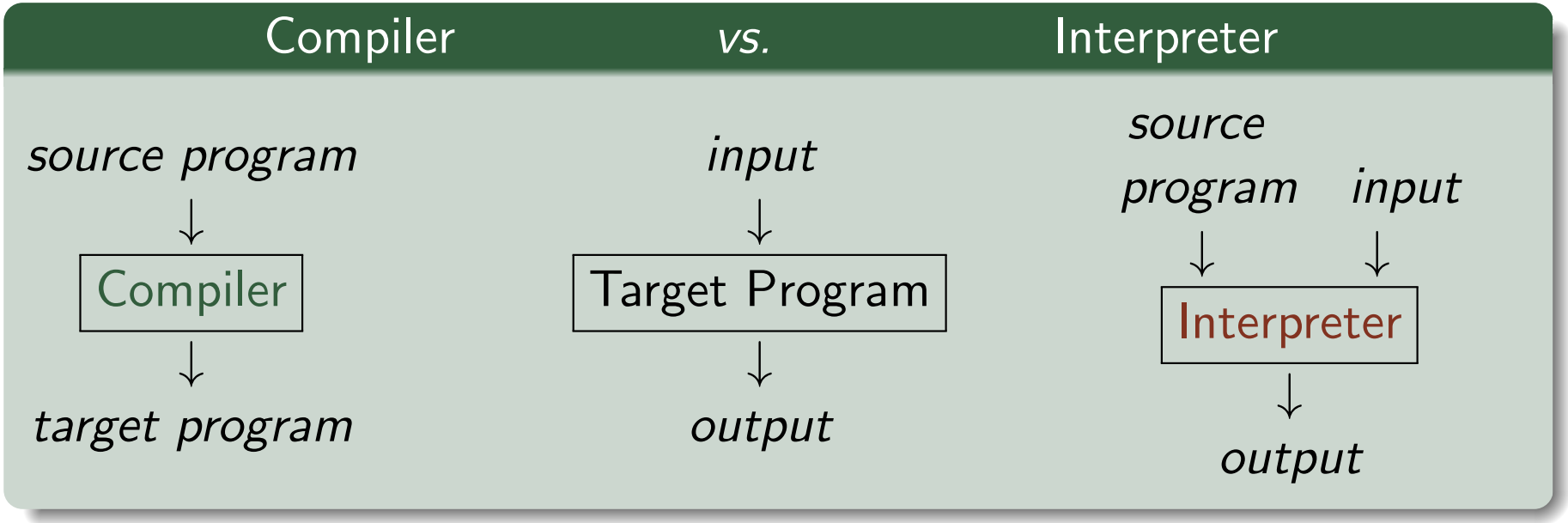
```
fun empty() = []

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?



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 & Strategy 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 a boolean.
In Paladim we also have array values.
An AbSyn value "knows" its type.

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

Interpreting 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* **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 integers*) *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 values of the same basic type*) *then* ($v_1 = v_2$) *else* **error**()
...	

Interpreting Expressions (Part 2)

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
...	
`let` **id** = Exp_1 `in` Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, name(\textbf{id}), v_1)$ $Eval_{Exp}(Exp_2, vtable', ftable)$
`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* ($v_1 = \textbf{true}$) *then* $Eval_{Exp}(Exp_2, vtable, ftable)$ *else* $Eval_{Exp}(Exp_3, vtable, ftable)$ *else* **error**()
id (*Exps*)	$def = lookup(ftable, name(\textbf{id}))$ *if* (def = *unbound*) *then* **error**() *else* $args = Eval_{Exps}(Exps, vtable, ftable)$ $Call_{Fun}(def, args, ftable)$

Intuitively, Exps can be represented as a list of expressions & $Eval_{Exps}$ evaluates a list of expressions, e.g., `map` $Eval_{Exp}$.

$Call_{Fun}$, introduced later, interprets a function call & returns a value.

Using SML facilities to Interpret Complex Exps

How to support the following language extension (?)

- $map : (\alpha \to \beta) * Array\ \alpha \quad \to \quad Array\ \beta$
- $map(f, \{x_1, x_2, \ldots, x_n\}) = \{f(x_1), f(x_2), \ldots, f(x_n))\}$
- and an array value is represented as an SML list of values.

Using SML facilities to Interpret Complex Exps

How to support the following language extension (?)

- $map : (\alpha \to \beta) * Array\ \alpha \quad \to \quad Array\ \beta$
- $map(f, \{x_1, x_2, \ldots, x_n\}) = \{f(x_1), f(x_2), \ldots, f(x_n)\}$
- and an array value is represented as an SML list of values.

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
...	
`map(` **id**, Exp^{arr} `)`	$arr = Eval_{Exp}(Exp^{arr}, vtable, ftable)$ $fdcl = lookup(\ ftable,\ name(\textbf{id})\ )$ *if (fdcl = unbound) then* **error**() *else if (arr is an array value)* *then map (fn x ⇒ $Call_{Fun}$(fdcl,[x],ftable)) arr* *else* **error**()

Function-Call Interpretation

- create a *new vtable* by binding the formal to the (already evaluated, i.e., values) 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

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

$Build_{ftable}(Funs) =$ `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`	
$Type$ `fid` ($TypeIds$) = Exp	`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?

Call-by-Value/Reference/Value Result Example

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

```
void main() {
    int x = 5;
    f(x, x);
    print(x);
}
```

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

Call-by-Value/Reference/Value Result Example

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

```
void main() {
    int x = 5;
    f(x, x);
    print(x);
}
```

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

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.

Call-by-Name Example

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

```
void f(int a, int b) {
    if(a > 0) print(a);
    else      print(b);
}
```

```
//infinite recursion
int g(int a) { return g(a); }

void main() { f(1, g(1));   }
```

Call-by-Name Example

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

```
void f(int a, int b) {
    if(a > 0) print(a);
    else      print(b);
}
```

```
//infinite recursion
int g(int a) { return g(a); }

void main() { f(1, g(1));   }
```

Differ when the actual parameters are not expressions:

Call by Name: 1.

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

Scopes: Static (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: Static (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: Static (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. Static scoping typically refers to the closest visible binding of a name!

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?

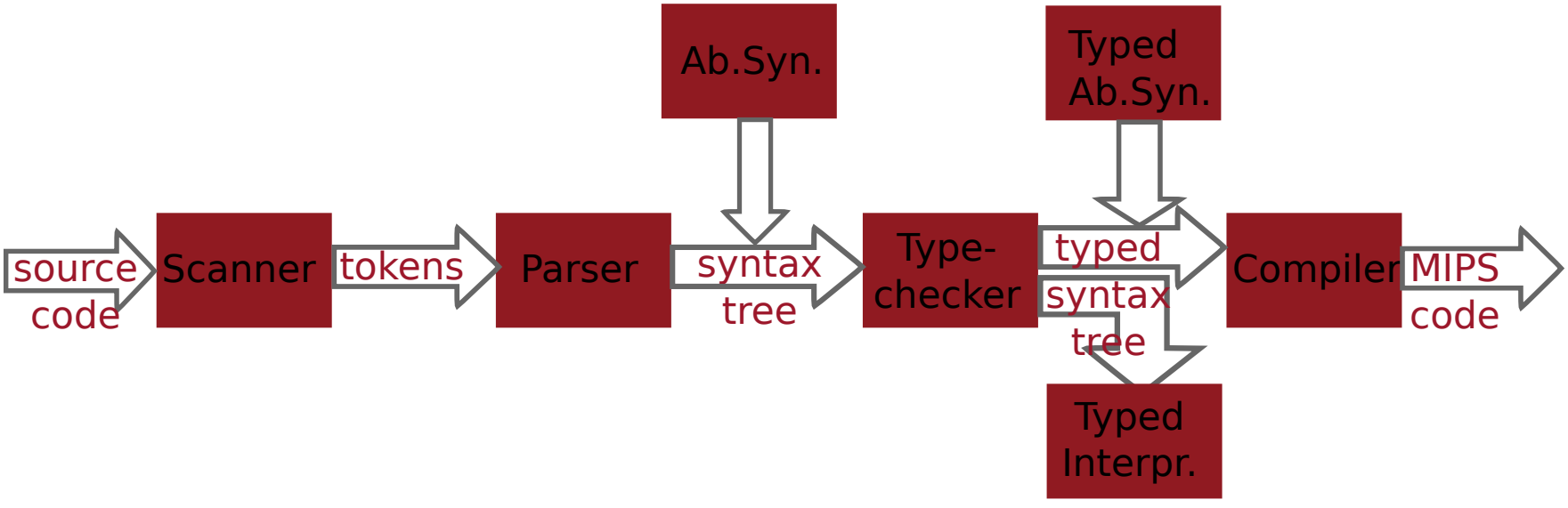
```
int x = 1;

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.

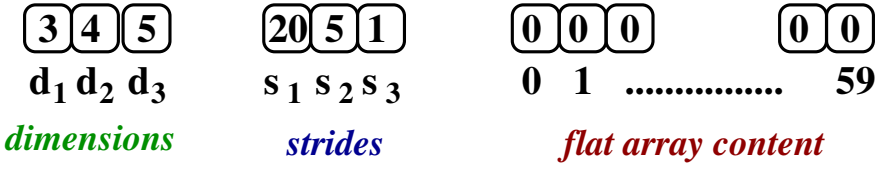
Compiler Modules of the Paladim Compiler



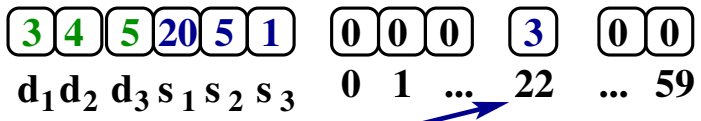
- The implementation of the parser is task 1 in G-Assignment (but a hand-implemented parser is provided to you).
- The parser produces an (untyped) AbSyn, i.e., file `AbSyn.sml`.
- The type checker receives an untyped AbSyn and produces a typed one, denoted TpAbSyn, in file `TpAbSyn.sml`, in which the type of any value, exp, stmt, can be querried via `typeOf*`.
- The program interpretation is performed on the TpAbSyn.

Interpreting Paladim: Types & Values in TpAbSyn

```
a : array of array of array of int;
a := new(3, 4, 5);

datatype Value = ...
  | Arr of  BasicVal array *
int list *  int list *  Type
```




```
a[1,0,2] := 3;
```




```
index_flat = 1 * 20 + 0 * 5 + 2 = 22
index_ok = (1 >= 0 && 1 < 3) &&
           (0 >= 0 && 0 < 4) &&
           (2 >= 0 && 2 < 5) = true
```

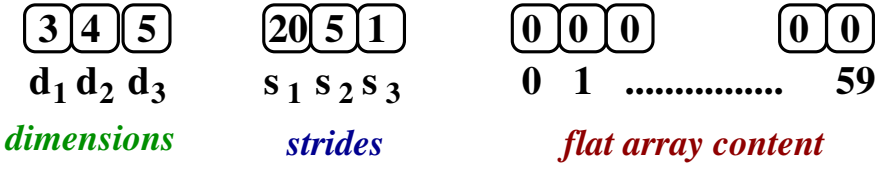
Paladim Types & Values

```
datatype BasicType = Int
  |                  Bool
  |                  Char

and Type =
      BType of       BasicType
  |   Array of int * BasicType
    (* int is the array's rank *)
```

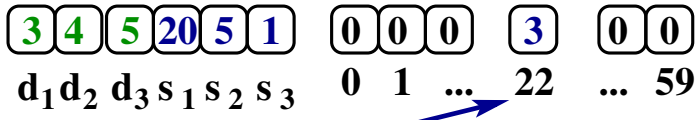
```
datatype BasicVal = Num of int  (*123*)
  | Log   of bool | Chr of char (*'c'*)

datatype Value = BVal of BasicVal
  | Arr of BasicVal array
                  * int list (* dims *)
                  * int list (* strides *)
                  * Type (* array's type*)
```

Interpreting Paladim: Types & Values in TpAbSyn

```
a : array of array of array of int;
a := new(3, 4, 5);

datatype Value = ...
  | Arr of  BasicVal array *
int list *  int list *  Type
```




```
a[1,0,2] := 3;
```




```
index_flat = 1 * 20 + 0 * 5 + 2 = 22
index_ok = (1 >= 0 && 1 < 3) &&
           (0 >= 0 && 0 < 4) &&
           (2 >= 0 && 2 < 5) = true
```

Making a New Array

```
mkNewArr( btp : BasicType, shpval : Value list, pos : Pos ) : Value =
    let val shape  = map ( fn d => case d of
                                     BVal(Num n) => n
                                   | _=>raise Error("dim size not int!",pos)
                         ) shpval
        val strides= mkStrides shape
        val arr_sz = foldl (op * ) 1 shape
        val arr    = Array.tabulate( arr_sz, fn i => getBasicVal0 btp )
        in  Arr( arr, shape, strides, Array(length shpval, btp) )
        end
```

VTABLE & Example of Paladim Program

Function Computing the First n Fibonnacci Numbers

```
function fillFib(n : int) : array of int
var r : array of int;
    i : int;
begin
   r    := new(n+1);
   i    := 1;
   r[0] := 0;  r[1] := 1;
   while (i < n) do
      begin
         i    := i + 1;
         r[i] := r[i-1] + r[i-2];
      end;
   return r;
end;
```

- Paladim is imperative, e.g., `i` takes different values inside loop.
- It follows that `vtable : (string * TpAbSyn.Value ref)`

Interpreting a Paladim Program

Interpretation Entry Point

```
fun execPgm(funlst : FunDec list) =
        (* ftab : (string * TpAbSyn.FunDec) list *)
        let val ftab  = buildFtab funlst
            val mainf = SymTab.lookup "main" ftab
        in  case mainf of
              NONE   => raise Error("in evalPgm: Undefined Main! ", (0,0))
            | SOME m => callFun(getFunDec m, [], [], [], ftab, (0,0))
end
```

- `buildFtab` is similar to the definition in the slides;
- `ftab` associates a fun/proc name with a fun/proc declaration;
- program execution starts with a call to procedure main that receives no parameters;
- if procedure `main` not found then error.
- getFunDec(f : FunDec) : (Type option * string * Dec list * StmtBlock * Pos) i.e., return type, function name, arguments' name & type, fun/proc body, postion is defined in file `TpAbSyn.sml`

Interpreting a Function Call – Implementation

TASK 5 requires you implement call-by-value-result for procedures.

`rtp: Type option, fid: string, fargs: Dec list, body: StmtBlock, pdcl: Pos`

Interpreting a Function Call & Helper Function "bindTypeIds"

```
fun callFun (
  (rtp, fid, fargs, body, pdcl),
  aargs : Value list, aexps, vtab,
  ftab, pcall ) : Value option =
        (* build args' SymTab *)
let val nvtab=bindTypeIds
                (fargs, aargs,..)
        (* eval fun's body *)
    val res  =execBlock
                (body,nvtab,ftab)
in  case (rtp, res) of
      (NONE  , _      ) => NONE
(*Procedure, modify ^ TASK 5.*)
    | (SOME t,SOME r) =>
      if typesEqual(t, typeOfVal r)
      then SOME r
      else raise Error(
           "ret type error", pcall)
    |_=>raise Error("...",pcall)end
```

```
(* NOTE THAT VTAB associates a var name *)
(*                  with a Value REFERENCE *)
(*imperative lang ⇒ destructive updates*)
(* vtab : (string * Value ref)            *)
(* callFun implements call by value       *)
fun bindTypeIds([],[],..) = SymTab.empty()
  | bindTypeids([],aa,..) =raise Error(..)
  | bindTypeIds(bb,[],..) =raise Error(..)
  | bindTypeIds(
        (Dec((faid,fatp), fap)) :: fargs,
        a::aargs, .. ) =
let val vtab = bindTypeIds(fargs,aargs,..)
in if( typesEqual( fatp, typeOfVal a ) )
   then case SymTab.lookup faid vtab of
         NONE   => SymTab.bind
                        faid (ref a) vtab
       | SOME m => raise Error
                ("duplicate arg name!",..)
   else raise Error("illegal arg type",..)end
```

Interpreting a Block of Statements

How are Return Statements Treated?

```
and execDecs ( [] ) = []
  | execDecs ( (Dec((id,tp),pos)::decs) ) : (string * Value ref) list =
      let val vtab = execDecs decs
          val newv = case tp of
                       Array(btp,_)=> Arr( Array.fromList [], [], [], tp)
                     | BType btp   => BVal( getBasicVal0 btp )
      in  SymTab.insert id (ref newv) vtab
      end
and execBlock( Block(decs, stmts) : StmtBlock, vtab, ftab ) : Value option =
      let val new_vtab = vtab @ execDecs(decs)
          val res      = foldl (fn (s, rr : Value option) =>
                                  case rr of
                                    NONE   => execStmt(s, new_vtab, ftab)
                                  | SOME r => SOME r
                               ) NONE stmts
      in  res  end
```

Interpreting a Block of Statements

How are Return Statements Treated?

```
and execDecs ( [] ) = []
  | execDecs ( (Dec((id,tp),pos)::decs) ) : (string * Value ref) list =
      let val vtab = execDecs decs
          val newv = case tp of
                       Array(btp,_)=> Arr( Array.fromList [], [], [], tp)
                     | BType btp   => BVal( getBasicVal0 btp )
      in  SymTab.insert id (ref newv) vtab
      end
and execBlock( Block(decs, stmts) : StmtBlock, vtab, ftab ) : Value option =
      let val new_vtab = vtab @ execDecs(decs)
          val res      = foldl (fn (s, rr : Value option) =>
                                  case rr of
                                    NONE   => execStmt(s, new_vtab, ftab)
                                  | SOME r => SOME r
                               ) NONE stmts
      in  res  end
```

- block's declared variables added to `vtable` (call to execDecs)
- *statements are executed as long as no return statement was reached, indicated by* `rr` *being* `NONE`. (See next slide!)

Interpreting Return and Assign Statement

execStmt returns NONE, with the exception of return statements:

```
fun execStmt ( Return   (SOME e, _), vtab, ftab ) =
            SOME ( evalExp(e, vtab, ftab) )
  | execStmt ( Assign( Var (id, _), e, pos), vtab, ftab ) =
      ( case SymTab.lookup id vtab of
          SOME vref => let val vnew = evalExp(e, vtab, ftab)
                       in  ( vref := vnew; NONE ) end
        | NONE => raise Error( "Variable Is Not In Symbol Table!", pos ) )
  | execStmt ( Assign( Index((id,_), inds), e, pos), vtab, ftab ) =
      ( case SymTab.lookup id vtab of
          SOME (ref arr ) =>
                let val indvals = map ( fn x => evalExp(x, vtab, ftab) ) inds
                    val ()=setArrIndex(arr, indvals, evalExp(e,vtab,ftab), pos)
                in  NONE end
        | NONE   => raise Error( "Array Not In Symbol Table!", pos ) )
```

- `a := exp;` evaluates exp to `vnew` and updates the symbol table value associated with a to `vnew`.
- `a[inds] := exp;` (i) finds the array value, `arr` of a from vtable, (ii) evaluates `inds` to a list of int values `indvals`, and (iii) updates the corresponding index to exp's evaluated value.

Interpreting If and While Statements

execStmt returns NONE, with the exception of return statements:

```
| execStmt ( IfThEl(cond, then_stmts, else_stmts, pos), vtab, ftab ) =
    ( case evalExp(cond, vtab, ftab) of
        BVal(Log true ) => execBlock(then_stmts, vtab, ftab)
      | BVal(Log false) => execBlock(else_stmts, vtab, ftab)
      | otherwise => raise Error("condition does not evaluate to bool", pos) )
| execStmt ( While(cond, body, pos), vtab, ftab ) =
    ( case evalExp(cond, vtab, ftab) of
        BVal(Log true ) =>
          ( case execBlock(body, vtab, ftab) of
              NONE      => execStmt ( While(cond, body, pos), vtab, ftab )
            | SOME v    => SOME v )
      | BVal(Log false) => NONE
      | otherwise => raise Error("condition does not evaluate to bool", pos) )
```

- if cond evaluates to `true` then the statement in the `then` block are executed, otherwise the ones in the `else` block
- if a loop condition evaluates to true, then the loop's body is executed, and execStmt is called (tail) recursively.

Interpreting Expressions

evalExp returns a Value

```
(* evalExp (e : Exp, vtab : (string * Value ref) list,
                      ftab : (string * FunDec   ) list ) : Value *)
| evalExp ( Plus(e1, e2, pos), vtab, ftab ) =
    let val res1   = evalExp(e1, vtab, ftab)
        val res2   = evalExp(e2, vtab, ftab)
    in  evalBinop(op +, res1, res2, pos)
    end

| evalExp ( FunApp( (fid, (atps,SOME rtp)), aargs, pos ), vtab, ftab ) =
    let val evargs = map (fn e => evalExp(e, vtab, ftab) ) aargs
        val resopt =
            if( fid = "read" orelse ... )
            then ... (* treatment of special functions *)
            else ( case ( SymTab.lookup fid ftab ) of
                      SOME f=>callFun(getFunDec f, evargs,aargs,vtab,ftab,pos)
                    | NONE  =>raise Error("Function Not In SymTab,", pos)
    in  case resopt of
          SOME v => v   (* functions always return a value *)
        | NONE   => raise Error("Function with no Result!", pos)
    end
```