UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Compilers: Interpretation

Alessandra Di Pierro
alessandra.dipierro@univr.it

Department of Computer Science
University of Verona

1. Intuition: Working with Abstract Syntax Trees and Symbol Tables
   - Example of an Abstract Syntax Tree
   - Simple Interpretation
   - Symbol Tables
   - Interpretation with Symbol Tables
   - Type Errors
   - Interpreting Function Calls

Simple Arithmetic Expressions - grammar

Let us consider a simple language of arithmetic expressions:

Exp $::=$ **numeric constant** (e.g $1, 42, 1337$)

Simple Arithmetic Expressions - grammar

Let us consider a simple language of arithmetic expressions:

$$\begin{array}{lcl} \text{Exp} & ::= & \textbf{numeric constant} \ (\text{e.g } 1, 42, 1337) \\ & | & \text{Exp + Exp} \end{array}$$

Simple Arithmetic Expressions - grammar

Let us consider a simple language of arithmetic expressions:

$$
\begin{array}{lcl}
\text{Exp} & ::= & \textbf{numeric constant} \ (\text{e.g } 1, 42, 1337) \\
 & | & \text{Exp + Exp} \\
 & | & \text{Exp – Exp} \\
 & | & \text{Exp} * \text{Exp} \\
 & | & \text{Exp / Exp}
\end{array}
$$

Simple Arithmetic Expressions - grammar

Let us consider a simple language of arithmetic expressions:

$$
\begin{array}{lll}
\text{Exp} & ::= & \textbf{numeric constant } (\text{e.g } 1, 42, 1337) \\
& | & \text{Exp + Exp} \\
& | & \text{Exp – Exp} \\
& | & \text{Exp} * \text{Exp} \\
& | & \text{Exp / Exp} \\
& | & \text{(Exp)}
\end{array}
$$

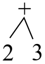
- Grouping (parentheses) are used to override usual operator priority - just as in ordinary mathematics.
- ...but they have no real *meaning* apart from their notation convenience.

Simple Arithmetic Expressions - AST

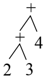
In the Abstract Syntax Tree (AST), we cover only the *essentials*.

`2` $\sim$ 2

`2 + 3` $\sim$ 

`(2 + 3)` $\sim$ 

`2 + 3 + 4` $\sim$ 

`(2 + 3) + 4` $\sim$ 

`2 + (3 + 4)` $\sim$ 

`2 + 3 * 4` $\sim$ 

Simple Arithmetic Expressions - AST in SML

If we want to work with these expressions, we will need to store them as a data structure in some other language. Often, we call the language we are manipulating the *object language*, and the language we are using the *implementation language* (or *meta language*). The ML in SML was originally for *Meta Language*!

Simple Arithmetic Expressions - AST in SML

If we want to work with these expressions, we will need to store them as a data structure in some other language. Often, we call the language we are manipulating the *object language*, and the language we are using the *implementation language* (or *meta language*). The ML in SML was originally for *Meta Language*!
We can define an AST type for our expression language like this:

```
datatype Exp = Constant of int
             | Plus of Exp * Exp
             | Minus of Exp * Exp
             | Times of Exp * Exp
             | Divide of Exp * Exp
```

```
datatype Exp = Constant of int
             | Plus of Exp * Exp
             | Minus of Exp * Exp
             | Times of Exp * Exp
             | Divide of Exp * Exp
```

We can now manually type in some SML values corresponding to arithmetic expressions:

`2`	$\sim$	`Constant 2`
`2 + 3`	$\sim$	`Plus(Constant 2, Constant 3)`
`(2 + 3)`	$\sim$	`Plus(Constant 2, Constant 3)`
`2 + 3 + 4`	$\sim$	`Plus(Plus(Constant 2, Constant 3), Constant 4)`
`(2 + 3) + 4`	$\sim$	`Plus(Plus(Constant 2, Constant 2), Constant 2)`
`2 + (3 + 4)`	$\sim$	`Plus(Constant 2, Plus(Constant 3, Constant 4))`
`2 + 3 * 4`	$\sim$	`Plus(Constant 2, Times(Constant 3, Constant 4))`

Parsing allows us to go from text to ASTs but, for now, we will type them in manually.

Simple Arithmetic Expressions - interpretation

We will define (in SML) an evaluation function for our language of simple arithmetic expressions:

```
eval :  Exp -> int
```

The AST type definition is recursive. Hence, the function is also written as a recursive function over the input tree.

Simple Arithmetic Expressions - interpretation

We will define (in SML) an evaluation function for our language of simple arithmetic expressions:

```
eval :  Exp -> int
```

The AST type definition is recursive. Hence, the function is also written as a recursive function over the input tree.

```
fun eval (Constant x)       = x
  | eval (Plus (e1, e2))    = eval e1 + eval e2
  | eval (Minus (e1, e2))   = eval e1 - eval e2
  | eval (Times (e1, e2))   = eval e1 * eval e2
  | eval (Divide (e1, e2))  = eval e1 div eval e2
```

And that is all it takes.

Jazzing up the language - variable bindings

We add another syntactic construct, namely `let`-expressions:

$$
\begin{array}{lcl}
\text{Exp} & ::= & \ldots \\
& \mid & \texttt{let}\ v\ \texttt{=}\ \text{Exp}\ \texttt{in}\ \text{Exp} \\
& \mid & v
\end{array}
$$

So we can now write

```
let x = 2*3 in x + x
```

Jazzing up the language - variable bindings

We add another syntactic construct, namely `let`-expressions:

$$\begin{array}{lcl} \text{Exp} & ::= & \ldots \\ & | & \texttt{let}\ v = \text{Exp}\ \texttt{in}\ \text{Exp} \\ & | & v \end{array}$$

So we can now write

```
let x = 2*3 in x + x
```

Or we would if we had a parser. The SML type now looks like this:

```
datatype Exp = Constant of int
             | Plus of Exp * Exp
             | Minus of Exp * Exp
             | Times of Exp * Exp
             | Divide of Exp * Exp
             | Let of (string * Exp * Exp)
             | Var of string
```

Jazzing up the language - interpreting variable bindings

We try to extend the evaluation function, but run into trouble:

```
...
  | eval (Var v)            = lookup value of v?
  | eval (Let (v, e1, e2)) = bind v to result of e1?
```

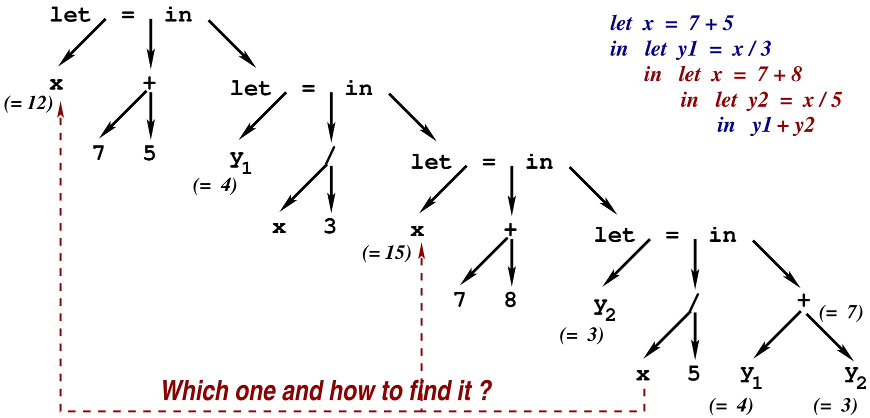
Jazzing up the language - interpreting variable bindings

We try to extend the evaluation function, but run into trouble:

```
...
  | eval (Var v)             = lookup value of v?
  | eval (Let (v, e1, e2)) = bind v to result of e1?
```

We need some data structure for keeping track of in-scope variables and their values.
Enter: symbol tables!

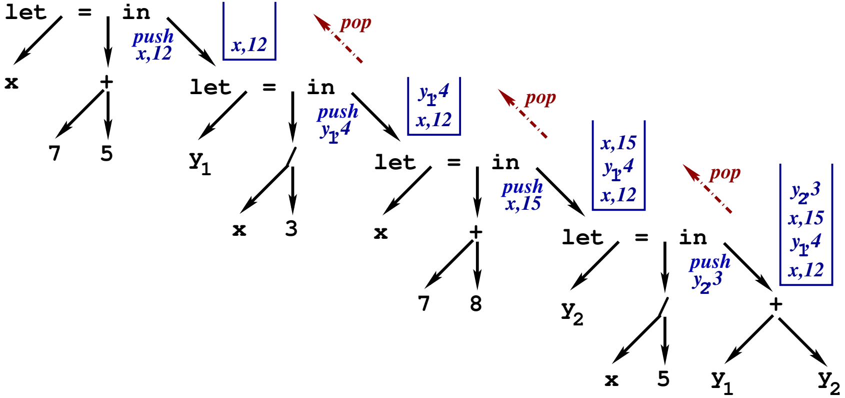
Symbol Tables in Theory



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?

Symbol Tables in Theory



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 Tables in Practice

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 Tables For Our Interpreter

We associate variable names with their integer value.

```
type SymTab = (string * int) list
```

- *empty:*

  ```
  fun empty () = []
  ```

- *bind:*

  ```
  fun bind k v vtable = (k,v) :: vtable
  ```

- *look-up:*

  ```
  fun lookup k0 ((k1,v) :: vtable) =
        if k0 = k1 then SOME v else lookup k0 vtable
    | lookup _ _ = NONE
  ```

- *enters* a new scope: call `bind` and recurse with new symbol table
- *exits* a scope: implicit when recursion ends.

Interpretation with Variable Bindings

We modify our evaluation function to have a new type:

```
eval :  SymTab -> Exp -> int
```

The previous cases now have to pass along a symbol table.

```
fun eval vtable (Constant x)      =
      x
  | eval vtable (Plus (e1, e2))   =
      eval vtable e1 + eval vtable e2
  | eval vtable (Minus (e1, e2))  =
      eval vtable e1 - eval vtable e2
  | eval vtable (Times (e1, e2))  =
      eval vtable e1 * eval vtable e2
  | eval vtable (Divide (e1, e2)) =
      eval vtable e1 div eval vtable e2
```

Interpretation with Variable Bindings

Only the new cases actually use the symbol table:

```
| eval vtable (Var v) =
    (case lookup v vtable of
        NONE => raise Fail ("Unknown variable " ^ v)
      | SOME x => x)
| eval vtable (Let (v, e1, e2)) =
    let val x = eval vtable e1
        val vtable' = bind v x vtable
    in eval vtable' e2 end
```

Jazzing up the language - more values!

We add another syntactic construct, namely `if-then-else`-expressions, the concept of a *boolean value*, and some new operators.:

Exp	::=	...
	|	`if` Exp `then` Exp `else` Exp
	|	**boolean constant** (i.e. true or false)
	|	Exp < Exp
	|	Exp = Exp
	|	Exp `and` Exp
	|	Exp `or` Exp

So we can now write

```
if 2 < 1 or 2 < 3 then 42 else 3
```

Modifying the AST with new value types

We can no longer assume that all variables are bound to integers - hence, we introduce a new type for *values*:

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

Modifying the AST with new value types

We can no longer assume that all variables are bound to integers - hence, we introduce a new type for *values*:

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

We also have to modify the `Exp` constructor `Constant`:

```
datatype Exp = Constant of Value
             | ...
```

Modifying the AST with new value types

We can no longer assume that all variables are bound to integers - hence, we introduce a new type for *values*:

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

We also have to modify the `Exp` constructor `Constant`:

```
datatype Exp = Constant of Value
             | ...
```

And the `SymTab` definition:

```
type SymTab = (string * Value) list
```

Type Error at Runtime

What if we are asked to evaluate `true + 2`? What should that result in? We are writing the interpreter, so we have to decide!

Type Error at Runtime

What if we are asked to evaluate `true + 2`? What should that result in? We are writing the interpreter, so we have to decide!
We can modify all the arithmetic cases as follows:

```
| eval vtable (Plus (e1, e2))    =
  (case (eval vtable e1, eval vtable e2) of
       (IntVal x, IntVal y) => IntVal (x + y)
     | _ => raise Fail "Operands to + are not integers")
```

I chose to consider boolean operands to arithmetic operators as an error, but as a language implementer, you can make a different choice. This requires taste and a sense of responsibility, and easily goes wrong.

Comparisons

Implementing comparisons is just like implementing arithmetic operators.

```
| eval vtable (Equal (e1, e2)) =
  (case (eval vtable e1, eval vtable e2) of
       (IntVal x, IntVal y) => BoolVal (x = y)
     | (BoolVal x, BoolVal y) => BoolVal (x = y)
     | _ => raise Fail "Operands to = are not of the same
| eval vtable (Less (e1, e2)) =
  (case (eval vtable e1, eval vtable e2) of
       (IntVal x, IntVal y) => BoolVal (x < y)
     | _ => raise Fail "Operands to < are not integers")
```

Note that I chose to permit comparisons of both booleans and integers - but the operands have to be of the same type! The result is always a boolean. Admittance to < is only for integers.

Branches

Branching is also quite straightforward. Is this correct?

```
| eval vtable (If (cond, truebranch, falsebranch)) =
  let val cond_res = eval vtable cond
      val truebranch_res = eval vtable truebranch
      val falsebranch_res = eval vtable falsebranch
  in (case cond_res of
          BoolVal true  => truebranch_res
        | BoolVal false => falsebranch_res
        | _ => raise Fail
               "Condition in if is not a boolean")
  end
```

Branches

Branching is also quite straightforward. Is this correct?

```
| eval vtable (If (cond, truebranch, falsebranch)) =
  let val cond_res = eval vtable cond
      val truebranch_res = eval vtable truebranch
      val falsebranch_res = eval vtable falsebranch
  in (case cond_res of
          BoolVal true  => truebranch_res
        | BoolVal false => falsebranch_res
        | _ => raise Fail
                "Condition in if is not a boolean")
  end
```

No!

Consider `if true then 2 else 2/0`. We definitely don't want to evaluate 2/0 unless we have to!

Branches

Better: we are careful not to evaluate the branches unless we have to:

```
| eval vtable (If (cond, truebranch, falsebranch)) =
  (case eval vtable cond of
       BoolVal true  => eval vtable truebranch
     | BoolVal false => eval vtable falsebranch
     | _ => raise Fail
            "Condition in if is not a boolean")
```

Same thing goes for `and` and `or` if we want them to be short-circuiting.

Short-circuiting and/or

```
| eval vtable (And (e1, e2)) =
  (case eval vtable e1 of
       BoolVal true  => eval vtable e2
     | BoolVal false => BoolVal false
     | _ => raise Fail "Operand to and is not a boolean")
| eval vtable (Or (e1, e2)) =
  (case eval vtable e1 of
       BoolVal true  => BoolVal true
     | BoolVal false => eval vtable e2
     | _ => raise Fail "Operand to or is not a boolean")
```

Testing the interpreter

```
- eval (empty ()) (Plus (Constant (IntVal 2),
                         Constant (IntVal 3)));
> val it = IntVal 5 : Value
```

Testing the interpreter

```
- eval (empty ()) (Plus (Constant (IntVal 2),
                         Constant (IntVal 3)));
> val it = IntVal 5 : Value

- eval (empty ()) (If (Constant (BoolVal true),
                       Constant (IntVal 2),
                       Divide (Constant (IntVal 2),
                               Constant (IntVal 0))));
> val it = IntVal 2 : Value
```

Seems to work...

Interpreting Larger Languages

We have been interpreting a very simple language. In particular, it has no functions. In larger languages, including the one (Fasto) you will be working on for the group project, we need another symbol table: the *function table*.

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

Interpreting Larger Languages

We have been interpreting a very simple language. In particular, it has no functions. In larger languages, including the one (Fasto) you will be working on for the group project, we need another symbol table: the *function table*.

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

Let us add functions to our language of arithmetic expressions:

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

datatype 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.

Generalising the Symbol Table

We need two symbol tables: a *variable table*, and a *function table*. Both map `strings` to information, so we parametrise on the type of information:

```
type 'info SymTab = (string * 'info) list
type VarTab = Value SymTab
type FunTab = FunDec SymTab
```

We do not have to modify the symbol table functions.

Function-Call Interpretation

We add another constructor to `Exp`:

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

Function-Call Interpretation

We add another constructor to Exp:

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

And 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:

```
fun eval vtable ftable (Constant x)      = x
  | eval vtable ftable (Plus (e1, e2))   =
    (case (eval vtable ftable e1, eval vtable ftable e2) of
         (IntVal x, IntVal y) => IntVal (x + y)
       | _ => raise Fail "Operands to + are not integers")
  | ...
```

You get the picture.

Function-Call Interpretation

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

```
| eval vtable ftable (Call (fname, args)) =
  let val vals =
        map (fn arg => eval vtable ftable arg) args
  in case lookup fname ftable of
         NONE => raise Fail ("Unknown function " ^ fname)
       | SOME fundec => callFun ftable fundec vals
  end
```

The `callFun` function is going to take care of the actual function-evaluation part.

Evaluating a Function

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

Evaluating a Function

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

```
and callFun ftable fundec args =
  case fundec of
      FunDec (rettype, fname, params, body) =>
      let val vtable = bindParams params args
          val result = eval vtable ftable body
      in case (result, rettype) of
             (IntVal x, Int) => IntVal x
           | (BoolVal x, Bool) => BoolVal x
           | _ => raise Fail "Return value mismatch"
      end
```

Note that a function is evaluated with a brand new variable table.

Binding Function Parameters

The bindParams function is conceptually simple, and is mostly error detection.

```
fun bindParams [] [] = empty ()
  | bindParams ((param_type, param_name) :: params) (v::vs)
    let val vtable = bindParams params vs
    in case (param_type, v) of
           (Int, IntVal x)   =>
             bind param_name (IntVal x) vtable
         | (Bool, BoolVal x) =>
             bind param_name (BoolVal x) vtable
         | _ => raise Fail "Parameter type mismatch"
    end
  | bindParams _ _ =
    raise Fail "Argument count mismatch"
```

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`:

```
fun makeFunTable [] = empty ()
  | makeFunTable (fundec::funs) =
    let val vtable = makeFunTable funs
    in case fundec of
           FunDec (rettype, fname, params, body) =>
           insert fname fundec vtable
    end
```

Missing error checking: multiple functions with the same name; 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:

```
fun evalProg prog =
  let val ftable = makeFunTable prog
  in case lookup "main" ftable of
          NONE => raise Fail "No main function defined"
        | SOME fundec => callFun ftable fundec []
  end
```

And It Works!

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

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