IPS: Course Organization & Introduction to AbSyn and Interpretation

Slides by Cosmin Oancea
cosmin.oancea@diku.dk

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

Department of Computer Science (DIKU)
University of Copenhagen

April 2023, IPS Lecture Slides

1. Course Organization
   - Content, Goals and Format of the Course
   - Weekly and Group Assignments, Exam
2. Compilation: From Source to Machine Code
   - Simple Introductory Example
   - Compilation Phases
3. Intuition: Working with Abstract Syntax Trees and Symbol Tables
   - Example of an Abstract Syntax Tree
   - Simple Interpretation
   - Symbol Tables
   - Interpretation with Symbol Tables
   - More Value Types

Learning Objectives of the Course

- Competences:
  - Design and implement a compiler: from a high-level language to machine code.
  - Evaluate and use appropriate tools and libraries for compilation.
  - Evaluate resource (time and space) consumption of the original and equivalent machine-code program.
- Skills:
  - Employ tools for lexical and syntactical analysis.
  - Describe and evaluate compiler development in written form.
  - Work with, and substantially extend, a non-trivial codebase.
- Knowledge:
  - Divide compilation into phases with their particular purpose.
  - Apply theoretical insight, e.g., regular expression, context-free grammars, graph coloring.
  - Explain how compiler tools work and their limitations.

Goals for Today's Lecture

- Course Organization:
  - weekly lectures and labs
  - weekly exercise session
  - weekly assignments, term project, and exam dates.
  - ... hopefully firmly back to pre-corona norms
- Birds-Eye View of the Compilation Phases
- How to Work With Abstract Syntax Trees (AbSyn or AST)
  - AbSyns are the main way to represent programs in a compiler
  - Today we aim to develop a view of how to work with them, enough for the weekly assignment and maybe the project.

Teaching format

- Please read full details on “Course Information” page in Absalon!
  - Including administrative formalia
- Two lecture sessions per week:
  - Scheduled for Mondays 13:15–15:00, Wednesdays 9:15–11:00.
  - Mainly slide-based presentation of material from course textbook and (where relevant) supplementary notes.
- (Up to) two “labs” per week:
  - Scheduled for the ~hour after each lecture
  - Less formal/structured presentations, e.g.:
    - Walk-through of relevant compiler code
    - Worked larger examples of hand-running algorithms
    - Repetition/elaboration of lecture material
- Based relatively closely on course textbook
  - Torben Æ. Mogensen. *Introduction to Compiler Design* (2nd ed.)
    - Get your copy ASAP!
  - Supplementary and background readings also listed on Absalon

Teaching format (continued)

- One exercise session (with TAs) per week:
  - Wednesdays 13:15–15:00
  - Discussion of small exercises (try to solve them in advance!)
  - Help with project and weekly assignments
  - Room allocations (all over North Campus) on Absalon
    - ... and in your myUCPH personal schedule (?)
- Absalon forum
  - Use as much as possible, to enable maximal information sharing
  - Try to follow (common-sense) usage suggestions on course info page, to help deal with expected large volume of posts
  - May organize into subforums as and when necessary; for now, just one big room.
- Discord chat forum:
  - Informal hangout, semi-regularly frequented by teaching staff.
  - Separated into channels and searchable, but...
    - still poorly suited for non-ephemeral questions and discussions persisting for more than a few hours.

Weekly Assignments, Group Project & Exam

- **Workload is front-loaded (first 5-6 weeks)!**
- Five Weekly Assignments
  - solved individually
  - need four approved assignments for admission to exam
  - initially published on Wednesdays, due next Wednesday (23:59)
  - first four can be resubmitted by the following week
- Group Project: ∼5 weeks implementation work + report.
  - solved and submitted in small groups (1–3 students, 2 is ideal)
    - but may discuss in general terms with others
  - needs to be approved for admission to exam
  - published on 26 April, due by 8 June
  - no resubmission possible, but use milestone handin on 26 May to get preliminary feedback
  - estimated 25–30 hours total workload; start early!
- Exam
  - as per course description: 4-hour written (ITX) test
  - may include questions relating to weekly assignments, project
  - administratively scheduled for 21 June, exact time/place TBA.

1. Course Organization
   - Content, Goals and Format of the Course
   - Weekly and Group Assignments, Exam
2. Compilation: From Source to Machine Code
   - Simple Introductory Example
   - Compilation Phases
3. Intuition: Working with Abstract Syntax Trees and Symbol Tables
   - Example of an Abstract Syntax Tree
   - Simple Interpretation
   - Symbol Tables
   - Interpretation with Symbol Tables
   - More Value Types

Compiler Phases

Pseudo-F# Code

```
let result =
  let x = 10 :: 20 :: 0x30 :: []
  List.map (fun a -> 2 * 2 * a) x
```

Discuss:

- what does the code do?
- write the (morally) equivalent C code
- what would be the necessary steps to translate the source code to machine code?

Translation from Source to C to Machine Code

Equivalent C Code

- initialize the array
- `map` translated to a loop
- some optimizations performed:
  2*2=4 at compile time,
  result reuses the space of `x`.

```
int x[3] = {10, 20, 0x30};
int i;
for (i = 0; i < 3; i++) {
  x[i] = 4*x[i];
}
```

Translation from Source to C to Machine Code

Equivalent C Code

- initialize the array
- map translated to a loop
- some optimizations performed:
  2*2=4 at compile time,
  result reuses the space of x.

```
int x[3] = {10, 20, 0x30};
int i;
for (i = 0; i < 3; i++) {
  x[i] = 4*x[i];
}
```

Generic RISC assembly code

- three-address-like code
- registers instead of variables
- explicit load and store ops
- shift instead of multiplication
- pointer arithmetic

```
load_imm   $2, 0      # $2 loop counter (i)
load_addr  $3, x_addr # $3 address of x
load_imm   $5, 3      # $5 loop bound (3)
loop:
    branch_ge  $2, $5, end
    load_word  $4, 0($3) # $4 holds x[i]
    shift_left $4, $4, 2 # multiply by 4
    store_word $4, 0($3) # store in x[i]
    add_imm    $3, $3, 4 # next x addr
    add_imm    $2, $2, 1 # i = i + 1
    jmp        loop
end:
```

Compiler Phases

- Front-End:
  - Lexical Analysis: split the program into individual tokens/lexemes; skip whitespace and comments; convert numerals to numbers ("0x30", "0048", ... $\leadsto$ 48; "0x3m"); recognize keywords
  - Syntactical Analysis: check cross-token well-formedness (matching parens, braces, etc.); construct tree representation of program.
  - Type Checking: check that vars declared before use, operators and functions invoked meaningfully ("`fun (x:int) -> y && x`")

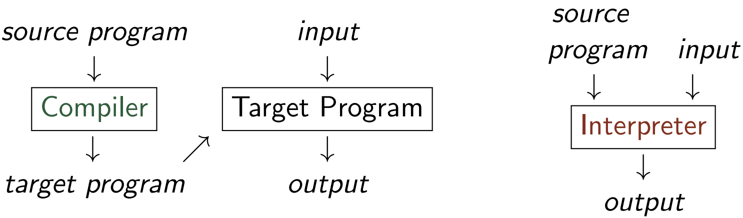
Compiler Phases

- Front-End:
  - Lexical Analysis: split the program into individual tokens/lexemes; skip whitespace and comments; convert numerals to numbers ("0x30", "0048", ... ⇝ 48; "0x3m"); recognize keywords
  - Syntactical Analysis: check cross-token well-formedness (matching parens, braces, etc.); construct tree representation of program.
  - Type Checking: check that vars declared before use, operators and functions invoked meaningfully ("`fun (x:int) -> y && x`")
- Intermediate-Language (IL) Code Generation & Optimizations:
  - IL and optimizations reused across several source languages.
  - Moving towards lower-level representations,
    e.g., how are arrays represented in memory,
  - but losing some high-level invariants (esp. type safety).

Compiler Phases

- Front-End:
  - Lexical Analysis: split the program into individual tokens/lexemes; skip whitespace and comments; convert numerals to numbers ("0x30", "0048", ... ⇝ 48; "0x3m"); recognize keywords
  - Syntactical Analysis: check cross-token well-formedness (matching parens, braces, etc.); construct tree representation of program.
  - Type Checking: check that vars declared before use, operators and functions invoked meaningfully ("`fun (x:int) -> y && x`")
- Intermediate-Language (IL) Code Generation & Optimizations:
  - IL and optimizations reused across several source languages.
  - Moving towards lower-level representations, e.g., how are arrays represented in memory,
  - but losing some high-level invariants (esp. type safety).
- Back-End: Generation of Machine-Code (MC)
  - How to translate IL constructs to MC, e.g., function calls?
  - What sequence of machine instrs best implements the IL code?
  - Machine resources are limited, e.g., arbitrary number of code variables need to be mapped to a finite number of registers.

Compilation vs. Interpretation



Compilation vs. Interpretation

source program ↓ $\boxed{\text{Compiler}}$ ↓ *target program* ↗ $\boxed{\text{Target Program}}$ (*input* ↓) ↓ *output*

source program ↓ *input* ↓ $\boxed{\text{Interpreter}}$ ↓ *output*

- Compilation results in a lower-level language program, e.g., machine code, which can be run on various inputs.
  Often substantially better performance ($\sim 10\times$ faster)
- Interpretation is good for impatient people:
  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.
  Often substantially easier to implement ($\sim 10\times$ faster?)

Structure of a Compiler

Program text
↓
Lexical analysis
↓
Symbol sequence
↓
Syntax analysis
↓
Syntax tree → Interpretation
↓
Typecheck
↓
(Annotated) 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

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:

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

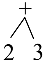
- Grouping (parentheses) are used to override usual operator priority - just as in ordinary mathematics.
- ...but they have no inherent *meaning* apart from indicating grouping.

Simple Arithmetic Expressions - AbSyn

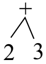
In the Abstract Syntax Tree (AbSyn), we cover only the *essentials*.

`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 - AbSyn in F#

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*).

Simple Arithmetic Expressions - AbSyn in F#

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*).
We can define an AbSyn type for our expression language like this:

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

```
type 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 F# 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))

Next lecture, we will discuss how to go from text strings to AbSyn values ("parsing") – for now, we will type them in manually.

Simple Arithmetic Expressions - interpretation

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

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

The AbSyn 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 F#) an evaluation function for our language of simple arithmetic expressions:

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

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

```
let rec eval e =
  match e with
    | Constant n      ->
```

Simple Arithmetic Expressions - interpretation

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

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

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

```
let rec eval e =
  match e with
    | Constant n       -> n
    | Plus   (e1, e2) ->
```

Simple Arithmetic Expressions - interpretation

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

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

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

```
let rec eval e =
  match e with
    | Constant n      -> n
    | Plus   (e1, e2) -> eval e1 + eval e2
    | Minus  (e1, e2) ->
```

Simple Arithmetic Expressions - interpretation

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

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

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

```
let rec eval e =
  match e with
    | Constant n      -> n
    | Plus   (e1, e2) -> eval e1 + eval e2
    | Minus  (e1, e2) -> eval e1 - eval e2
    | Times  (e1, e2) -> eval e1 * eval e2
    | Divide (e1, e2) -> eval e1 / eval e2
```

And that is all it takes.

Jazzing up the language - variable bindings

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

Exp	::=	...
	|	`let` **var** = Exp `in` Exp
	|	**var**

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:

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

So we can now write

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

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

```
type 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:

```
let rec eval e =
  match e with
    ...
    | eval (Var v)            -> // lookup value of v. where?
    | eval (Let (v, e1, e2)) -> // bind v to result of e1?
```

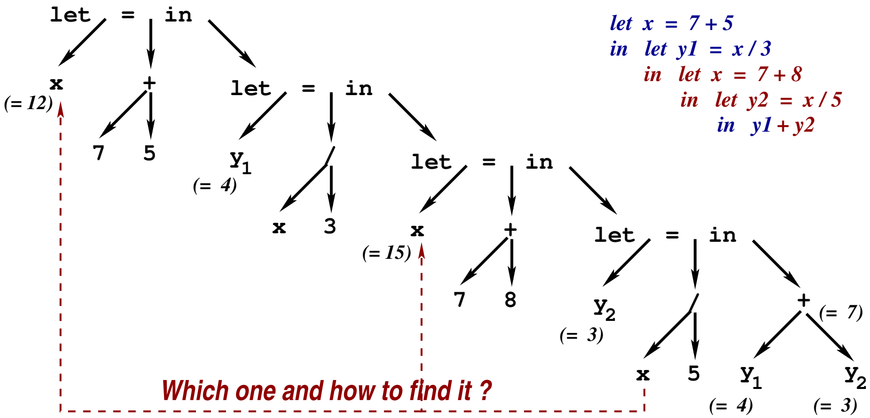
Jazzing up the language - interpreting variable bindings

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

```
let rec eval e =
  match e with
    ...
    | eval (Var v)              -> // lookup value of v. where?
    | 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!

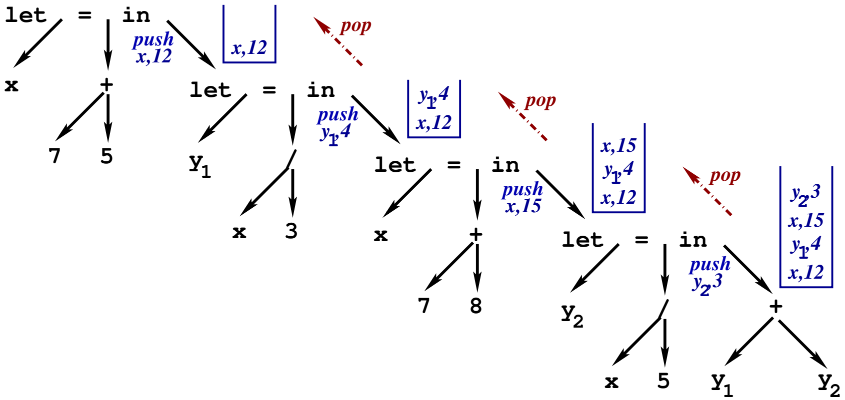
Symbol Tables in Theory



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 groups following bindings
- *exit*s a scope: restores the table to what it has been before entering the current scope.

Type of info associated with a variable depends on what we are doing

- Interpretation, type checking, code generation, optimization, ...

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

Symbol Tables For Our Interpreter

Associate variable names with their integer value using `SymTab<int>`.

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)
```

- **enters** a new scope: call `bind` and recurse with new symbol table
- **exits** a scope: implicit when recursion ends (because we use a purely-functional representation)

Interpretation with Variable Bindings

We modify our evaluation function to have a new type:

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

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

```
let rec eval vtable e =
  match e with
    | Constant n     -> n
    | Plus (e1, e2) ->
        (eval vtable e1) + (eval vtable e2)
    | Minus  (e1, e2) -> ...
    | Times  (e1, e2) -> ...
    | Divide (e1, e2) -> ...
```

Interpretation with Variable Bindings

Only the new cases actually use the symbol table:

```
exception MyError of string
let rec eval vtable e =
  match e with
    ...
    | Var v ->
      match lookup v vtable with
        | None -> raise (MyError ("Unknown variable: "+v))
        | Some n -> n

    | Let (v, e1, e2) ->
       // evaluate e1 in current vtable, giving n1
       // extend vtable with binding of v to n1
       // evaluate e2 in new vtable
```

Interpretation with Variable Bindings

Only the new cases actually use the symbol table:

```
exception MyError of string
let rec eval vtable e =
  match e with
    ...
    | Var v ->
      match lookup v vtable with
        | None -> raise (MyError ("Unknown variable: "+v))
        | Some n -> n

    | Let (v, e1, e2) ->
       // evaluate e1 in current vtable, giving n1
       // extend vtable with binding of v to n1
       // evaluate e2 in new vtable
       let n1 = eval vtable e1
       let vtable1 = bind v n1 vtable
       eval vtable1 e2
```

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 x < 1 or 10 < x then 42 else 3
```

Modifying the AbSyn with new value types

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

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

Modifying the AbSyn with new value types

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

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

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

```
type Exp = Constant of Value
         | ...
         | Equal of Exp * Exp
         | Less of Exp * Exp
         | If of Exp * Exp * Exp
         | And of Exp * Exp
         | Or of Exp * Exp
```

Modifying the AbSyn with new value types

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

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

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

```
type Exp = Constant of Value
         | ...
         | Equal of Exp * Exp
         | Less of Exp * Exp
         | If of Exp * Exp * Exp
         | And of Exp * Exp
         | Or of Exp * Exp
```

And we will use `SymTab<Value>` to bind a variable name to a `Value`

Type Error at Runtime

What if we are asked to evaluate `true + 2`? What should that result in? If we are the ones designing the language, we get to decide!

Type Error at Runtime

What if we are asked to evaluate `true + 2`? What should that result in? If we are the ones designing the language, we get to decide!
We have to modify all the arithmetic cases like so:

```
let rec eval vtable e =
  match e with
     ...
   | 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")
```

Here, we chose to consider boolean operands to arithmetic operators as an error, but as a language designer, you can do whatever you want. This requires taste and a sense of responsibility, and easily goes wrong. See: PHP, Javascript.

Comparisons

Implementing comparisons is just like implementing arithmetic operators.

```
let rec eval vtable e =
 match e with ...
  | Equal (e1, e2) ->
     match (eval vtable e1, eval vtable e2) with
      | (IntVal  n1, IntVal n2)  -> BoolVal (n1 = n2)
      | (BoolVal b1, BoolVal b2) -> BoolVal (b1 = b2)
      | _ -> raise (MyError "Operands to = not same type")
  | Less (e1, e2) ->
     match (eval vtable e1, eval vtable e2) with
      | (IntVal n1, IntVal n2) -> BoolVal (n1 < n2)
      | _ -> raise (MyError "Operands to < not integers")
```

Note that I chose to permit equality-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?

```
let rec eval vtable e =
  match e with
    ...
    | If (cond, truebranch, falsebranch) ->
      let cond_res        = eval vtable cond
      let truebranch_res  = eval vtable truebranch
      let falsebranch_res = eval vtable falsebranch
      match cond_res with
        | BoolVal true  -> truebranch_res
        | BoolVal false -> falsebranch_res
        | _ -> raise (MyError "If condition not a bool")
```

Branches

Branching is also quite straightforward. Is this correct?

```
let rec eval vtable e =
  match e with
    ...
    | If (cond, truebranch, falsebranch) ->
      let cond_res        = eval vtable cond
      let truebranch_res  = eval vtable truebranch
      let falsebranch_res = eval vtable falsebranch
      match cond_res with
        | BoolVal true  -> truebranch_res
        | BoolVal false -> falsebranch_res
        | _ -> raise (MyError "If condition not a bool")
```

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:

```
let rec eval vtable e =
  match e with
    ...
    | If (cond, truebranch, falsebranch) ->
       match eval vtable cond with
         | BoolVal true  -> eval vtable truebranch
         | BoolVal false -> eval vtable falsebranch
         | _ -> raise (MyError "If condition not a bool")
```

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

Short-circuiting and/or

```
let rec eval vtable e =
  match e with
     ...
    | And (e1, e2) ->
       match eval vtable e1 with
         | BoolVal true  -> eval vtable e2
         | BoolVal false -> BoolVal false
         | _ -> raise (MyError "And operand not a bool")
    | Or (e1, e2) ->
       match eval vtable e1 with
         | BoolVal true  -> BoolVal true
         | BoolVal false -> eval vtable e2
         | _ -> raise (MyError "Or operand not a bool")
```

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; ship it!

Coming up

- Wednesday 26/4:
  - Introduction to lexing and parsing
  - Presentation of Group Project and Weekly Assignment 1
    - Install .NET SDK 7.0 (which includes F#) before Wednesday, so that you are ready to go (or know what you need help with) at exercise sessions.
- Monday 1/5:
  - Interpretation, continued: functions, parameter passing, ...
  - Book notation for interpretation
- Wednesday 3/5 (with Robert):
  - Type checking / inference