Virtual Machines

Lecture 4 – Ocaml Language and Compilers and Interpreters

Overview

- Mutable features of Ocaml: Imperative programming
- Modular Programming- Modules
- Compilers and Interpreters - Introduction

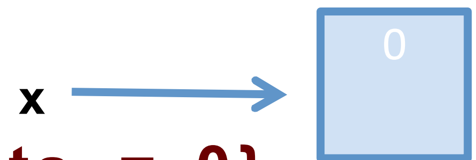
Imperative Programming

Mutable features of OCaml

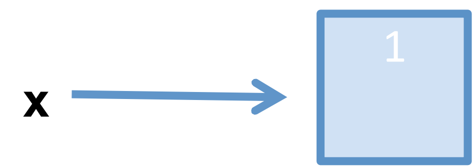
- Time to finally admit that OCaml has mutable features
  - It is not a ***pure language***
  - ***Pure*** = no side effects
- Sometimes it really is best to allow values to change, e.g.,
  - call a function that returns an incremented counter every time
  - efficient hash tables
- OCaml variables really are immutable
- But OCaml has mutable ***references, fields,*** and ***arrays…***

References

- aka “ref” or “ref cell”
- Pointer to a typed location in memory




```
# let x = ref 0;;
val x : int ref = {contents = 0}
# !x;;
- : int    = 0
# x:=1;;
- : unit    = ()
# !x;;
- : int = 1
```



References

- **The** binding of **`x`** to the pointer is immutable, as always
  - **`x`** will always point to the same location in memory
  - unless its binding is shadowed
- But the contents of the memory may change

Implementing a counter

```
let counter = ref 0
let next_val =
  fun () ->
    counter := (!counter) + 1;
    !counter
```

- **`next_val()`** returns **`1`**
- then **`next_val()`** returns **`2`**
- then **`next_val()`** returns **`3`**
- etc.

Implementing a counter

```
(* better *)
let next_val =
  let counter = ref 0 in fun () ->
    incr counter;
    !counter
```

Question

What's wrong with this implementation?

```
let next_val = fun () ->
  let counter = ref 0
  in incr counter;
     !counter
```

A. It won't compile, because `counter` isn't in scope in the final line
B. It returns a reference to an integer instead of an integer
C. It returns the wrong integer
D. Nothing is wrong
E. I don't know

Question

What's wrong with this implementation?

```
let next_val = fun () ->
  let counter = ref 0
  in incr counter;
     !counter
```

A. It won't compile, because `counter` isn't in scope in the final line
B. It returns a reference to an integer instead of an integer
C. It returns the wrong integer
D. Nothing is wrong
E. I don't know

Follow-up

Q: Why does this implementation work?

```
let next_val =
  let counter= ref 0 in fun () ->
    incr counter;
    !counter
```

A: the closure captures **`counter`** in its environment

References

- **Syntax:** `ref e`
- **Evaluation:**
  - Evaluate `e` to a value `v`
  - Allocate a new ***location*** `loc` in memory to hold `v`
  - Store `v` in `loc`
  - Return `loc`
  - Note: locations are first-class values; can pass and return from functions
-

References

- **Syntax:** `e1 := e2`
- **Evaluation:**
  - Evaluate **`e2`** to a value **`v2`**
  - Evaluate **`e1`** to a location **`loc`**
  - Store **`v2`** in **`loc`**
  - Return **`()`**
-

References

- **Syntax: `!e`**
  - note: not negation
- **Evaluation:**
  - Evaluate **`e`** to **`loc`**
  - Return contents of **`loc`**
-

References

- **Syntax:** **`e1; e2`**
- **Evaluation:**
  - Evaluate **`e1`** to a value **`v1`**
  - then **throw away** that value (note: **`e1`** could have side effects)
  - evaluate **`e2`** to a value **`v2`**
  - return **`v2`**
-

Implementing semicolon

Semicolon is essentially syntactic sugar:

```
e1; e2
(* means the same as *)
let () = e1 in e2
```

Except: suppose it's not the case that **`e1 : unit`**…

- let syntax: type error
- semicolon syntax: type warning

Question

What does **w** evaluate to?

```
let x = ref 42
let y = ref 42
let z = x
let () = x := 43
let w = (!y)+(!z)
```

A. 42
B. 84
C. 85
D. 86
E. None of the above

Aliases

References may have **aliases**:

```
let x = ref 42
let y = ref 42
let z = x
let () = x := 43
let w = (!y) +(!z)
```

z and **x** are aliases:

- in "**let z = x**", **x** evaluates to a location, and **z** is bound to the same location
- changing the contents of that location will cause both **!x** and **!z** to change

Equality

- Suppose we have two refs…
  - `let r1 = ref 3110`
  - `let r2 = ref 3110`
- Double equals is *physical equality*
  - `r1 == r1`
  - `r1 != r2`
- Single equals is *structural equality*
  - `r1 = r1`
  - `r1 = r2`
  - `ref 3110 <> ref 2110`
- **You usually want single equals**

Mutable fields

Fields of a record type can be declared as mutable:

```
# type point = {x:int; y:int; mutable c:string};;
type point = {x:int; y:int; mutable c:string; }
# let p = {x=0; y=0; c="red"};;
val p : point = {x=0; y=0; c="red"}
# p.c <- "white";;
- : unit = ()
# p;;
val p : point = {x=0; y=0; c="white"}
# p.x <- 3;;
Error: The record field x is not mutable
```

Implementing refs

Ref cells are essentially syntactic sugar:

```
type 'a ref = { mutable contents: 'a }
let ref x = { contents = x }
let ( ! ) r = r.contents
let ( := ) r newval = r.contents <- newval
```

- that type is declared in **`Pervasives`**
- the functions are compiled down to something equivalent

Benefits of immutability

- Programmer doesn't have to think about aliasing; can concentrate on other aspects of code
- Language implementation is free to use aliasing, which is cheap
- Often easier to reason about whether code is correct
- Perfec fit for concurrent programming
-

But there are downsides:

- I/O is fundamentally about mutation
- Some data structures (hash tables, arrays, …) hard(er) to implement in pure style

Try not to abuse your new-found power!

Arrays

Arrays generalize ref cells from a single mutable value to a sequence of mutable values

```
# let v = [|0.; 1.|];;
val v : float array = [|0.;1.|]
# v.(0) <- 5.;;
- : unit = ()
# v;;
- : float array = [|5.; 1.|]
```

Arrays

- **Syntax:** `[|e1; ...; en|]`
- **Evaluation:** evaluates to an **`n`**-element array, whose elements are initialized to **`v1...vn`**, where **`e1`** evaluates to **`v1`**, ...,**`en`** evaluates to **`vn`**
-

Arrays

- **Syntax:** `e1.(e2)`
- **Evaluation:** if `e1` evaluates to `v1`, and `e2` evaluates to `v2`, and `0<=v2<n`, where `n` is the length of array `v1`, then evaluates to element at offset `v2` of `v1`. If `v2<0` or `v2>=n`, raises `Invalid_argument`.
-

Arrays

- **Syntax:** `e1.(e2) <- e3`
  - **Evaluation:** if `e1` evaluates to `v1`, and `e2` evaluates to `v2`, and `0<=v2<n`, where `n` is the length of array `v1`, and `e3` evaluates to `v3`, then mutate element at offset `v2` of `v1` to be `v3`. If `v2<0` or `v2>=n`, raise `Invalid_argument`. Evaluates to `()`.

See `Array` module for more operations, including more ways to create arrays

Control structures

Traditional loop structures are useful with imperative features:

- `while e1 do e2 done`
- `for x=e1 to e2 do e3 done`
- `for x=e1 downto e2 do e3 done`

(they work like you expect)

Imperative Programming

https://realworldocaml.org/v1/en/html/index.html

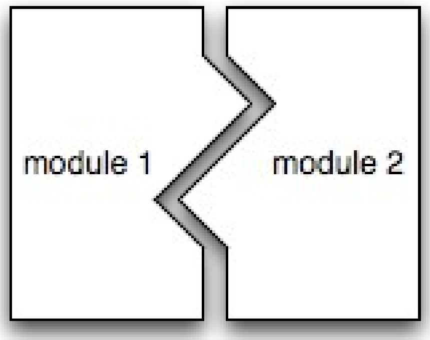
- Imperative programming(see attached Chapter8.pdf). Please look at all highlighted examples. Let's discuss now:
  - Laziness (pg 9)
  - Memoization (pg 10)
  - Order of evaluation (pg 19)
- File operations (see attached FileManipulation.pdf)

Modular Programming

Modularity

Modular programming: code comprises independent ***modules***

– developed separately
– understand behavior of module in isolation
– reason locally, not globally



Java features for modularity

- classes, packages
  - organize identifiers (classes, methods, fields, etc.) into namespaces
- interfaces
  - describe related classes
- public, protected, private
  - control what is visible outside a namespace
- subtyping, inheritance
  - enables code reuse

OCaml features for modularity

- structures
  - organize identifiers (functions, values, etc.) into namespaces
- signatures
  - describe related modules
- abstract types
  - control what is visible outside a namespace
- functors, includes
  - enable code reuse

...together, these features comprise the OCaml module system

Running examples
(see attached code modules.ml)

- Stacks
- Queues

- *Functional* aka *persistent* data structures:
  - never mutate the data structure
  - old versions of the data structure *persist* and are still usable

Stack module

```
module MyStack =struct
  type 'a stack=
  | Empty
  | Entry of 'a * 'a stack

  let empty = Empty
  let is_empty s = s = Empty
  let push x s = Entry (x, s)
  let peek = function
    | Empty -> failwith "Empty"
    | Entry(x,_) -> x
  let pop = function
    | Empty -> failwith "Empty"
    | Entry(_,s) -> s
end
```

Another Stack module

```
module ListStack = struct
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let peek = function
    | []     -> failwith "Empty"
    | x::_   -> x
  let pop = function
    | []     -> failwith "Empty"
    | _::xs  -> xs
end
```

Might seem backwards...

- In Java, might write

```
s = new Stack();
s.push(1);
s.pop();
```

- The stack is to the left of the dot, the method name is to the right
- In OCaml, it's seemingly backward:

```
let  s  =  MyStack.empty    in
let  s'  =  MyStack.push    1  s  in
let  one  =  MyStack.peek    s'
```

- the stack is an argument to every function (common **idioms** are last argument or first argument)
-

Yet another **Stack** module

Assume a type **'a fastlist** with constructor **FastNil** and **FastCons** that have a more efficient implementation than **'a list...**

```
module FastStack = struct
  let empty = FastNil
  ...
end
```

A multitude of implementations

- Each has its own *representation type*
  - **MyStack** uses **'a stack**
  - **ListStack** uses **'a list**
  - **FastStack** uses (hypothetical) **'a fastlist**
- Which causes each module to have a different *interface*…

Defining signatures

```
module type ListStackSig = sig
  val empty : 'a list
  val is_empty : 'a list -> bool
  val push : 'a -> 'a list -> 'a list
  val peek : 'a list -> 'a
  val pop : 'a list -> 'a list
end
module ListStack : ListStackSig = struct
  ...
end
```

Stack signatures

```
module ListStack : sig
  val empty : 'a list
  val is_empty : 'a list -> bool
  val push : 'a -> 'a list -> 'a list
  val peek : 'a list -> 'a
  val pop : 'a list -> 'a list
end

module MyStack : sig
  type 'a stack = Empty | Entry of 'a * 'a stack
  val empty : 'a stack
  val is_empty : 'a stack -> bool
  val push : 'a -> 'a stack -> 'a stack
  val peek : 'a stack -> 'a
  val pop : 'a stack -> 'a stack
end
```

A multitude of implementations

- Client code shouldn't **need to know** what the representation type is
- Client code shouldn't **get to know** what the representation type is
- Rule of thumb: clients *will exploit knowledge* of representation if you let them
  - One day a client of **ListStack** will write **x::s** instead of **push x s**
  - And the day you upgrade to fast lists, you will break their code
- So how can we unify these representations?

Abstract types

```
module type Stack = sig
  type 'a stack
  val empty     : 'a stack
  val is_empty  : 'a stack -> bool
  val push      : 'a -> 'a stack -> 'a stack
  val peek      : 'a stack -> 'a
  val pop       : 'a stack -> 'a stack
end
```

`'a stack` is **abstract**: signature declares only that type exists, but does not define what the type is

Abstract types

```
module MyStack : Stack = struct
  type 'a stack = type 'a stack = Empty | Entry of 'a * 'a stack
  . . .

module ListStack : Stack struct
  type ='a stack = 'a list
  . . .

module FastListStack : Stack = struct
  type 'a stack = 'a fastlist
  ...
```

- Every module of type **Stack** must define the abstract type
- Inside the module, types are synonyms
- Outside the module, types are not synonyms

  List.hd ListStack.empty will not compile

Abstract types

General principle: **information hiding** *aka* **encapsulation**

- *Clients* of **Stack** don't need to know it's implemented (e.g.) with a list
- *Implementers* of **Stack** might one day want to change the implementation
  - If list implementation is exposed, they can't without breaking all their clients' code
  - If list implementation is hidden, they can freely change

Abstract types

Common **idiom** is to call the abstract type **t** :

```
module type Stack = sig
  type 'a t
   val  empty     : 'a  t
   val  is_empty  : 'a  t  ->  bool
   val  push      : 'a  ->  'a  t  ->  'a  t
   val  peek      : 'a  t  ->  'a
   val  pop       : 'a  t  ->  'a  t
end

module ListStack : Stack = struct
   type 'a t  = 'a list
   ...
```

Queues

- Two implementations of functional queues in code accompanying lecture:
  - Queues as lists (poor performance)
  - Queues as two lists (good performance)
- See attached code modules.ml

Module syntax

```
module ModuleName [:t] = struct
  definitions
end
```

- the **ModuleName** must be capitalized
- type **t** (which must be a module type) is optional
- definitions can include **`let`**, **`type`**, **`exception`**
- definitions can even include nested **module**

A module creates a new **namespace:**

```
module M = struct let x = 42 end
let y = M.x
```

Signature syntax

```
module type SignatureName = sig
   type specifications
end
```

- type specifications aka *declarations*
- the **SignatureName** does not have to be capitalized but usually is
- declarations can include **`val`**, **`type`**, **`exception`**
  - `val name : type`
  - `type t [= definition]`
- declarations can even include nested **module type**

Type checking

If you give a module a type...

```
module Mod : Sig = struct ... end
```

then type checker ensures...

1. **Signature matching:** everything declared in `Sig` must be defined in `Mod`
2. **Encapsulation:** nothing other than what's declared in `Sig` can be accessed from outside `Mod`

1. Signature matching

```
module type S1 = sig
  val x: int
  val y: int
end
module M1 : S1 = struct
  let x = 42
end
(* type error:
   Signature mismatch:
   The value `y' is required but not provided
*)
```

2. Encapsulation

```
module type S2 = sig
  val x: int
end
module M2 : S2 = struct
  let x = 42
  let y = 7
end
M2. y
(* type error: Unbound value M2.y *)
```

Evaluation

To evaluate a module

```
  struct
    def1
    def2
    . . .
    defn
  end
```

evaluate each definition in order

Modules and files

Compilation unit = **`myfile.ml`** + **`myfile.mli`**

If **`myfile.ml`** has contents *DM*
[and **`myfile.mli`** has contents *DS*]
then OCaml compiler behaves essentially as though:

```
module Myfile [: sig DS end] =
struct
   DM
end
```

Modules and files

File
stack.mli:

```
(* The type of a stack whose
   elements are type 'a *)
type 'a t
(* The empty stack *)
val empty : 'a  t
(* Whether the  stack is empty*)
val is_empty :  'a t -> bool
(* [push x s] is the stack [s] with
   [x] pushed on the top *)
val push : 'a -> 'a t -> 'a t
(* [peek s] is the top element of [s].
   raises Failure if [s] is empty *)
val peek : 'a t -> 'a
(* [pop s] pops and discards the top
   element of [s].
   raises Failure if [s] is empty *)
val pop : 'a t -> 'a t
```

File
stack.ml:

```
(* Represent a stack as a list.
   [x::xs] is the stack with
   element [x]and top remaining
   elements [xs]. *)
type 'a t = 'a list
let empty = []
let is_empty s = s = []
let push x s   x :: s
(* Consider: using options
   instead of exceptions. *)
let peek = function
  | []   -> failwith "Empty"
  | x::_ -> x
let pop = function
  | []    -> failwith "Empty"
  | _::xs -> xs
```

Note: no **struct** or **sig** keywords, no naming of module or module type

Note: comments to client in **.mli**, comments to implementers in **.ml**

What about main()?

- there is no specific entry point into a module
- Common **idiom** is to make the last definition in a module be a function call that starts computation, e.g.
-

```
let _ = go_do_stuff ()
```

And you might call that function **main** instead of **go_do_stuff**, but you don't need to

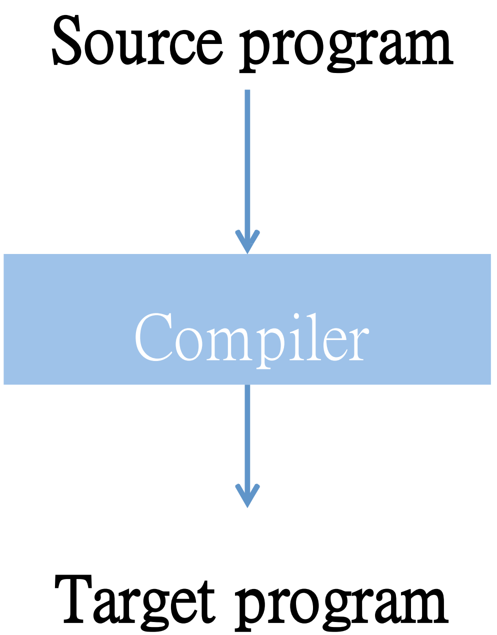
Ocaml language

https://realworldocaml.org/v1/en/html/index.html

- more about modules see in attached Chapter4.pdf

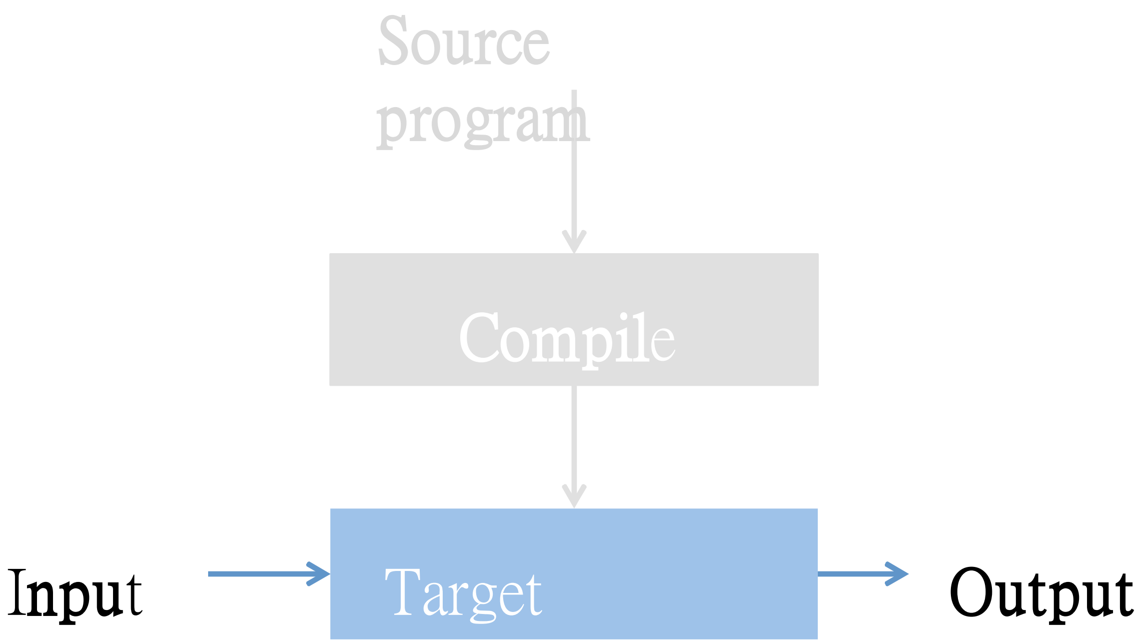
Compilers and Interpreters

Compilation



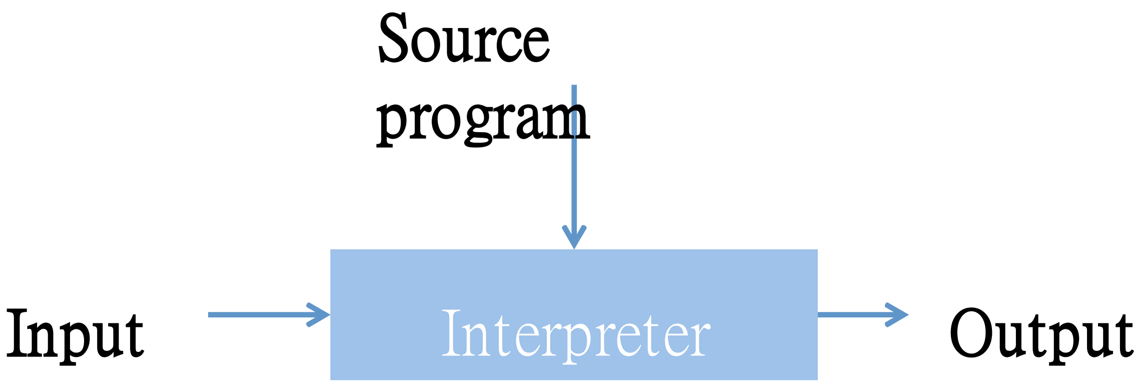
code as data: the compiler is code that operates on data; that data is itself code

Compilation



the compiler goes away; not needed to run the program

Interpretation

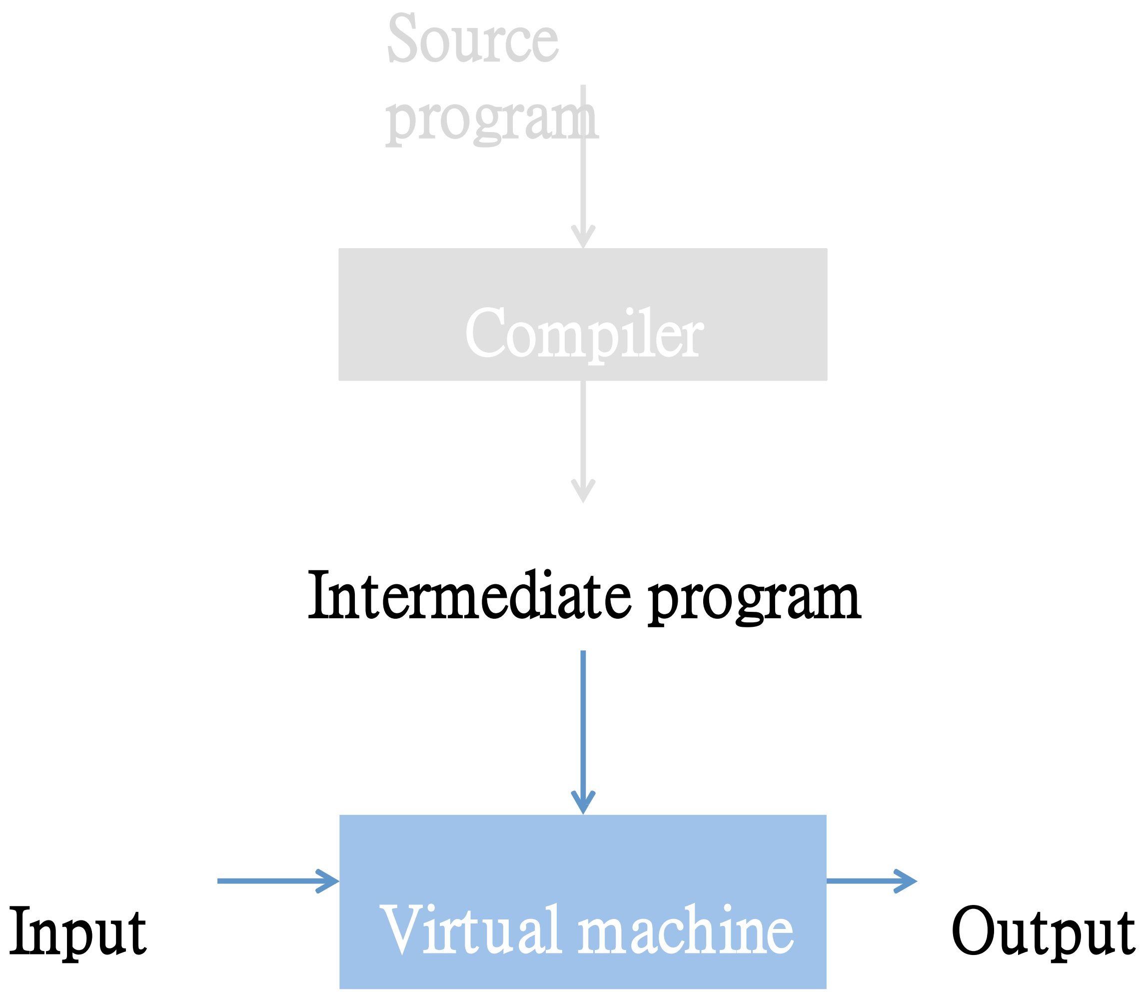


the interpreter stays; needed to run the program

Compilation vs. interpretation

- Compilers:
  - primary job is ***translation***
  - typically lead to better performance of program
- Interpreters:
  - primary job is ***execution***
  - typically lead to easier implementation of language
    - maybe better error messages and better debuggers

Mixed compilation and interpretation



the VM is the interpreter; needed to run the program; Java and OCaml can both work this way

Architecture

Architecture of a compiler is pipe and filter

- Compiler is one long chain of filters, which can be split into two phases
- **Front end:** translate source code into a tree data structure called ***abstract syntax tree*** (AST)
- **Back end:** translate AST into machine code
-

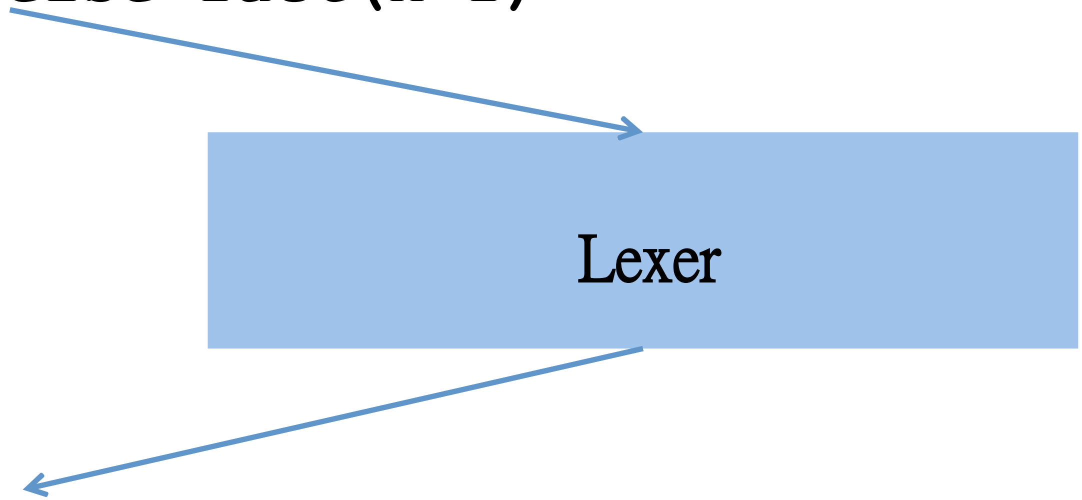
Front end of compilers and interpreters largely the same:

- *Lexical analysis* with lexer
- *Syntactic analysis* with parser
- *Semantic analysis*

Front end

Character stream:

```
if x=0 then  1 else fact(x-1)
```

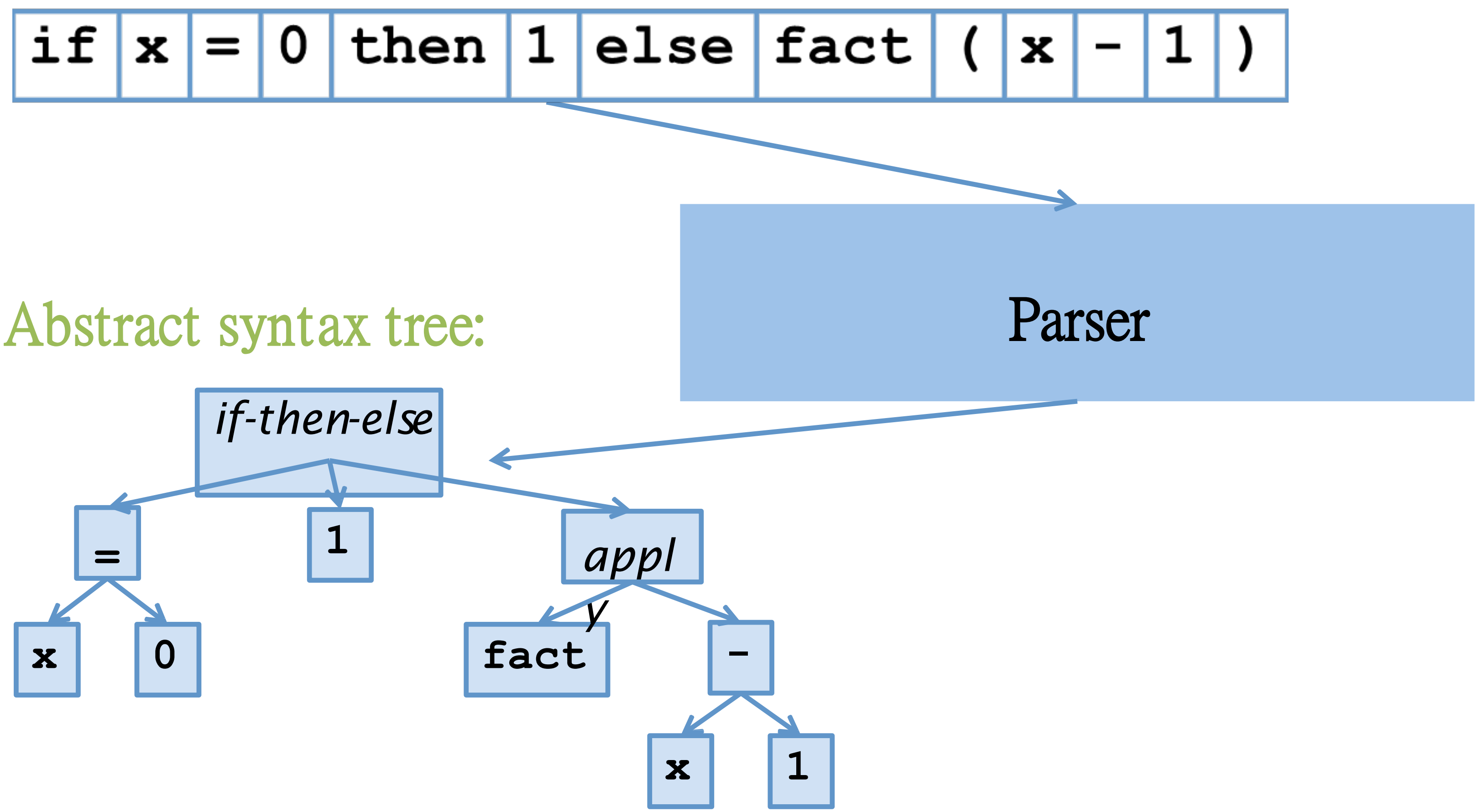


Token stream:

`if`	`x`	`=`	`0`	`then`	`1`	`else`	`fact`	`(`	`x`	`-`	`1`	`)`

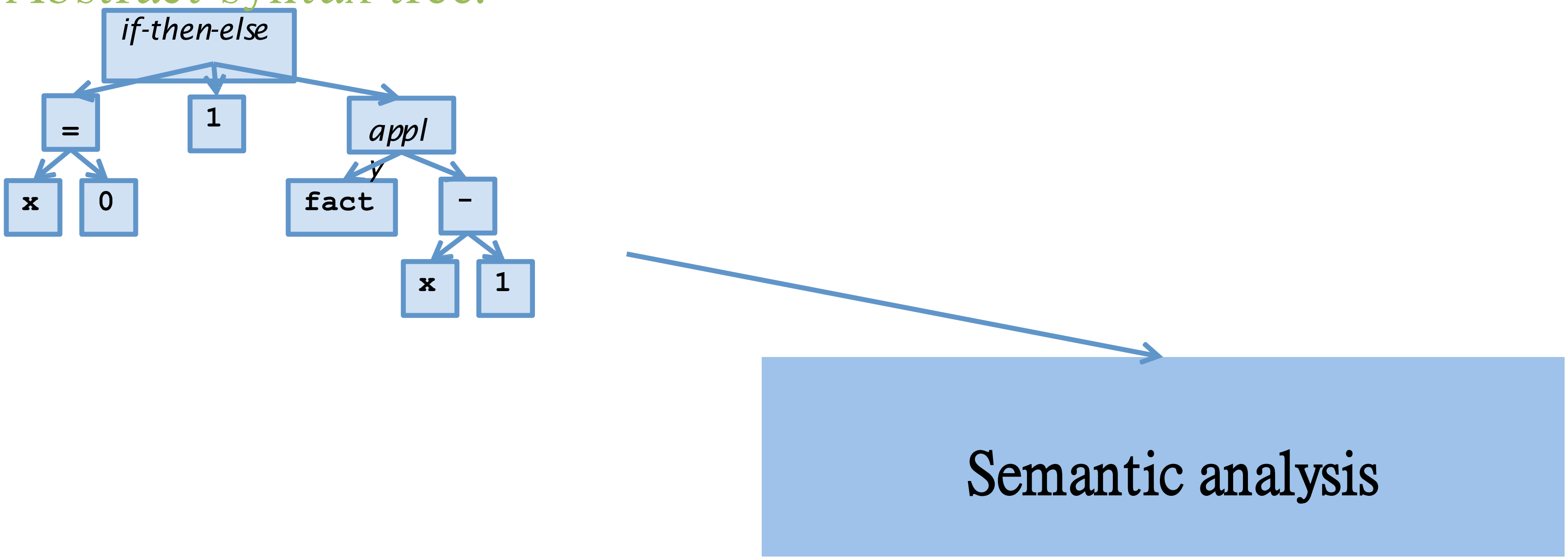
Front end

Token stream:



Front end

Abstract syntax tree:



- accept or reject program
- ***decorate*** AST with types
- etc.

After the front end

- **Interpreter** begins executing code using the abstract syntax tree (AST)
- **Compiler** begins translating code into machine language
  - Might involve translating AST into a simpler *intermediate representation* (IR)
  - Eventually produce *object code*

Implementation

Functional languages are well-suited to implement compilers and interpreters

- Tree data types
- Functions defined by pattern matching on trees
- Semantics leads naturally to implementation with functions

EXPRESSION INTERPRETER

Arithmetic expressions

Goal: write an interpreter for expressions involving integers and addition

Path to solution:

- let's assume lexing and parsing is already done
- need to take in AST and interpret it
- intuition:
  - an expression e takes a single ***step*** to a new expression e'
  - expression keeps stepping until it reaches a ***value***

Solution: see attached **`interp1.ml`**

Arithmetic expressions

Goal: extend interpreter to **`let`** expressions

Path to solution:

- extend AST with a variant for **`let`**
- add a branch to **`step`** to handle **`let`**
- that requires *substitution…*

`let` expressions

```
let x = e1 in e2
```

Evaluation:

- Evaluate **`e1`** to a value **`v1`**
- Substitute **`v1`** for **`x`** in **`e2`**, yielding a new expression **`e2'`**
- Evaluate **`e2'`** to **`v`**
- Result of evaluation is **`v`**

Arithmetic expressions

Goal: extend interpreter to **`let`** expressions

Path to solution:

- extend AST with a variant for **`let`**
- add a branch to **`step`** to handle **`let`**
- that requires ***substitution…***
- hence a substitution model interpreter

Solution: see attached **`interp2.ml`**

FORMAL SYNTAX

Abstract syntax of expression lang.

```
e ::= x | i | e+e
    | let x = e1 in e2
```

`e`, **`x`**, **`i`**: *meta-variables* that stand for pieces of syntax

- **`e`**: expressions
- **`x`**: program variables
- **`i`**: integers

`::=` and **`|`** are *meta-syntax*: used to describe syntax of language

notation is called *Backus-Naur Form* (BNF) from its use by Backus and Naur in their definition of Algol-60

Abstract syntax of expr. lang.

```
e ::= x | i | e+e
    | let x = e1 in e2
```

Note how closely the BNF resembles the OCaml variant we used to represent it!

Lab work

1. Try all the examples from the attached files and try to write your simple examples

2. Design a simple imperative language and write the types which correspond to the language AST (abstract syntax tree)