Programming Languages

Interpretation

Interpreters

What is an interpreter?

An interpreter is a program that executes programs.

It involves two programming languages:

Implementation language

The language in which the interpreter is defined.

Source language

The language in which the programs being interpreted are written.

Frequently asked question

Could these two languages coincide?

Yes, there can be interpreters capable of interpreting themselves.
This is not of particular interest in this course.

Languages we will use

In today's class we will define several interpreters.

Implementation language

We will define interpreters in Haskell.

Source language

We will define interpreters for different source languages (e.g., imperative and functional languages).

Note

We will work with minimalist ("toy") source languages.
They are sufficient to illustrate the important concepts.

Concrete syntax vs. abstract syntax

The interpreter receives as input data representing a program written in the source language.

How is a program represented?

Concrete syntax

A program can be represented as a **text string**.
For example:

```
"while (true) { x = x + 1; }" :: String
```

Abstract syntax

A program can be represented as a **syntax tree**.
For example:

```
EWhile
  ETrue
  (EAssign "x" (EAdd (EVar "x") (EConstNum 1)))
  :: Program
```

Concrete syntax vs. abstract syntax

In today's class

We will represent programs as abstract syntax trees.

Converting concrete syntax (text) into abstract syntax (tree) is a problem of **syntactic analysis**.

It is outside the scope of this course.

Arithmetic expression language

Consider a language of arithmetic expressions built inductively as follows:

1. An integer constant is an expression.
2. The sum of two expressions is an expression.

Expressions can be represented with a data type:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EAdd Expr Expr         -- e1 + e2
```

For example, "1 + (2 + 3)" is represented by:

```
EAdd (EConstNum 1)
     (EAdd (EConstNum 2) (EConstNum 3))
```

Exercise (1)

Define an interpreter:

```
eval :: Expr -> Int
```

Extension with boolean constants

Could we extend the language with boolean constants?

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EConstBool Bool        -- True, False
          | EAdd Expr Expr         -- e1 + e2
```

Problem

The interpreter is no longer a function `eval :: Expr -> Int`.

Values

We define a data type `Val` to represent *values*
(possible results of computations):

```
data Val = VN Int
         | VB Bool
```

Exercise (2)

Define an interpreter:

```
eval :: Expr -> Val
```

Local definitions and environments

We want to extend the language with local definitions:

```
let x = 3 in (let y = x + x in 1 + y)
```

We need to keep track of the value of each variable.

Environments

An **environment** is a dictionary that maps identifiers to values.

We will assume we have types:

`Id`	identifiers (variable names)
`(Env a)`	environments mapping identifiers to values of type `a`

and the following interface:

```
emptyEnv  :: Env a
lookupEnv :: Env a -> Id -> a
extendEnv :: Env a -> Id -> a -> Env a
```

Extension with variables and local definitions

We extend the expression language:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EConstBool Bool        -- True, False
          | EAdd Expr Expr         -- e1 + e2
          | EVar Id                -- x
          | ELet Id Expr Expr      -- let x = e1 in e2
```

Problem

What is the result of evaluating `(EVar "x")`?

The interpreter is no longer a function `eval :: Expr -> Val`.

Exercise (3)

Define an interpreter:

```
eval :: Expr -> Env Val -> Val
```

Extension with variables and local definitions

Comment

In the language with local declarations, an expression does not denote a value, but a *function* that returns a value depending on the given environment:

```
eval :: Expr -> (Env Val -> Val)
```

Introduction

Basic interpreters

Imperative features

Functional features

Effects

Interpreter with memory for an imperative language

We want to extend the language with imperative features:

1. Assignments: `x := e`
2. Sequential composition: `e1; e2`

We will assume that:

1. The *value* of an assignment is 0.
   (Not very important, just a convention).
2. The semantics of composition `e1; e2` corresponds to first evaluating `e1`, discarding its value, and then evaluating `e2`.

For example, the following program should result in the integer 4:

```
let x = 1 in x := x + 1; x + x
```

Interpreter with memory for an imperative language

Immutable variables

In a purely functional language, variables are *immutable*.

- The environment directly maps each variable to a value.

Mutable variables

In an imperative language, variables are typically *mutable*.

- The environment does **not** map each variable to a value.
- The environment maps each variable to a **memory address**.
- Additionally, there is a **memory** that maps addresses to values.
- Evaluating a program can modify the memory.

Interpreter with memory for an imperative language

Memories

A **memory** is a dictionary that maps addresses to values.

We will assume we have types:

`Addr`	memory addresses
`(Mem a)`	memories mapping addresses to values of type `a`

and the following interface:

```
emptyMem     :: Mem a
freeAddress  :: Mem a -> Addr
load         :: Mem a -> Addr -> a
store        :: Mem a -> Addr -> a -> Mem a
```

Extension with assignment and sequential composition

We extend the language with imperative features:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EConstBool Bool        -- True, False
          | EAdd Expr Expr         -- e1 + e2
          | EVar Id                -- x
          | ELet Id Expr Expr      -- let x = e1 in e2
          | ESeq Expr Expr         -- e1; e2
          | EAssign Id Expr        -- x := e
```

Problem

The result of evaluating an assignment (`x := e`) cannot be just a value (consider e.g., `let x = 1 in x := 2; x`).

What should the type of the interpreter be?

Exercise (4)

Define an interpreter:

```
eval :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
```

Extension with control structures

Now let's extend the interpreter with control structures:

```
data Expr =
    EConstNum Int          -- 1, 2, 3, ...
  | EConstBool Bool        -- True, False
  | EAdd Expr Expr         -- e1 + e2
  | EVar Id                -- x
  | ELet Id Expr Expr      -- let x = e1 in e2
  | ESeq Expr Expr         -- e1; e2
  | EAssign Id Expr        -- x := e
  | ELtNum Expr Expr       -- e1 < e2
  | EIf Expr Expr Expr     -- if e1 then e2 else e3
  | EWhile Expr Expr       -- while e1 do e2
```

Exercise (5)

Define an interpreter:

```
eval :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
```

Introduction

Basic interpreters

Imperative features

Functional features

Effects

Interpreters for functional languages

Almost all functional languages are based on the λ-**calculus**.

The λ-calculus is a language that has only three constructs:

```
data Expr = EVar Id              -- x
          | ELam Id Expr         -- \ x -> e
          | EApp Expr Expr       -- e1 e2
```

It is possible to program using only these constructs.

But we will extend the λ-calculus to make it more comfortable and resemble a realistic language.

Interpreters for functional languages

What will be the result of evaluating `(\ x -> x + x)`?

We need to extend the type of values to include functions.

First attempt

The value of a function is its "source code".

```
data Val = VN Int
         | VB Bool
         | VFunction Id Expr
```

For example, the result of evaluating `(\ x -> x + x)` would be:

```
VFunction "x" (EAdd (EVar "x") (EVar "x"))
```

We will see shortly that this is not correct.

Functional interpreter (first attempt)

Consider the λ-calculus extended with integers and booleans:

```
data Expr =
    EVar Id                  -- x
  | ELam Id Expr             -- \ x -> e
  | EApp Expr Expr           -- e1 e2
  | EConstNum Int            -- 1, 2, 3, ...
  | EConstBool Bool          -- True, False
  | EAdd Expr Expr           -- e1 + e2
  | ELet Id Expr Expr        -- let x = e1 in e2
  | EIf Expr Expr Expr       -- if e1 then e2 else e3
```

Exercise (6)

Define an interpreter:

```
eval :: Expr -> Env Val -> Val
```

Functional interpreter (first attempt)

Exercise

Evaluate the following program with the newly defined interpreter:

```
let sum = \ x -> \ y -> x + y in
let f = sum 5 in
let x = 0 in
  f 3
```

Problem: variable capture

We would expect the result to be 8 but it is 3.

The problem is that the variable `f` is bound to the value:

```
VFunction "y" (EAdd (EVar "x") (EVar "y"))
```

The variable `x` is **free**.

Functional interpreter (second attempt: with closures)

Second attempt

The value of a function is a **closure**.

A closure is a value that includes:

1. The source code of the function.
2. An environment that gives values to all its free variables.

```
data Val = VN Int
         | VB Bool
         | VClosure Id Expr (Env Val)
```

Functional interpreter (second attempt: with closures)

Recall the λ-calculus expressions with integers and booleans:

```
data Expr =
    EVar Id                  -- x
  | ELam Id Expr             -- \ x -> e
  | EApp Expr Expr           -- e1 e2
  | EConstNum Int            -- 1, 2, 3, ...
  | EConstBool Bool          -- True, False
  | EAdd Expr Expr           -- e1 + e2
  | ELet Id Expr Expr        -- let x = e1 in e2
  | EIf Expr Expr Expr       -- if e1 then e2 else e3
```

Exercise (7)

Define an interpreter using closures:

```
eval :: Expr -> Env Val -> Val
```

Evaluation strategies

There are different techniques for evaluating an application `(e1 e2)`.

These techniques are known as **evaluation strategies**.

The interpreter we just made uses the following strategy:

1. **Call-by-value**:
   Evaluate `e1` until it becomes a closure.
   Evaluate `e2` until it becomes a value.
   Proceed with evaluating the body of the function.
   The parameter is bound to the value of `e2`.

There are other evaluation strategies; for example:

2. **Call-by-name**:
   Evaluate `e1` until it becomes a closure.
   Proceed directly to evaluate the body of the function.
   The parameter is bound to the unevaluated expression `e2`.
   Each time the parameter is used, the expression `e2` is evaluated.
3. **Call-by-need**: (We will see it shortly).

Call-by-name interpreter

In the call-by-name evaluation strategy:

- When evaluating `(let x = e1 in e2)`, `e2` is evaluated directly.
  The variable `x` is bound to an unevaluated copy of `e1`.
- Environments do **not** map identifiers to values.
  Environments map identifiers to **thunks**.

Thunks

A *thunk* is data that includes:

1. An unevaluated expression.
2. An environment that gives values to all its free variables.

Thunks and values are defined as follows:

```
data Thunk = TT Expr (Env Thunk)
data Val = VN Int
         | VB Bool
         | VClosure Id Expr (Env Thunk)
```

Call-by-name interpreter

Recall the λ-calculus expressions with integers and booleans:

```
data Expr =
    EVar Id                  -- x
  | ELam Id Expr             -- \ x -> e
  | EApp Expr Expr           -- e1 e2
  | EConstNum Int            -- 1, 2, 3, ...
  | EConstBool Bool          -- True, False
  | EAdd Expr Expr           -- e1 + e2
  | ELet Id Expr Expr        -- let x = e1 in e2
  | EIf Expr Expr Expr       -- if e1 then e2 else e3
```

Exercise (8)

Define an interpreter that uses the call-by-name strategy:

```
eval :: Expr -> Env Thunk -> Val
```

Call-by-need interpreter

Call-by-need.

To evaluate an application `(e1 e2)`:

- Evaluate `e1` until it becomes a closure.
- Proceed directly to evaluate the body of the function.
- The parameter is bound to the unevaluated expression `e2`.
- The first time the parameter is needed, `e2` is evaluated.
- The result is saved to avoid evaluating `e2` again.

For this, we need **mutable memory**.

Call-by-need interpreter

Values

In `call-by-need` there are two types of values:

1. Final values (integers, booleans, closures, etc.).
2. Values pending evaluation (*thunks*).

```
data Val = VN Int
         | VB Bool
         | VClosure Id Expr (Env Addr)
         | VThunk Expr (Env Addr)
```

Properties of call-by-need evaluation

1. The environment maps identifiers to addresses.
2. The memory maps addresses to final values or thunks.
3. The result of evaluating an expression is a final value.

Call-by-need interpreter

Recall the λ-calculus expressions with integers and booleans:

```
data Expr =
    EVar Id                  -- x
  | ELam Id Expr             -- \ x -> e
  | EApp Expr Expr           -- e1 e2
  | EConstNum Int            -- 1, 2, 3, ...
  | EConstBool Bool          -- True, False
  | EAdd Expr Expr           -- e1 + e2
  | ELet Id Expr Expr        -- let x = e1 in e2
  | EIf Expr Expr Expr       -- if e1 then e2 else e3
```

Exercise (9)

Define an interpreter that uses the call-by-need strategy:

```
eval :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
```

Introduction

Basic interpreters

Imperative features

Functional features

Effects

Error propagation and handling

We extend the arithmetic expression language:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EAdd Expr Expr         -- e1 + e2
          | EVar Id                -- x
          | ELet Id Expr Expr      -- let x = e1 in e2
          | EDiv Expr Expr         -- e1 / e2
          | ETry Expr Expr         -- try e1 else e2
```

- The division operator fails if the divisor is 0.
- The control structure (`try e1 else e2`) evaluates `e1` and, in case of failure, proceeds to evaluate `e2`.

How is the type of the interpreter modified?

Exercise (10)

Define an interpreter:

```
eval :: Expr -> Env Int -> Maybe Int
```

Non-determinism

We extend the arithmetic expression language:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EAdd Expr Expr         -- e1 + e2
          | EVar Id                -- x
          | ELet Id Expr Expr      -- let x = e1 in e2
          | EAmb Expr Expr         -- amb e1 e2
```

- The operator `(amb e1 e2)` chooses between `e1` and `e2` in a non-deterministic way.

How is the type of the interpreter modified?

Exercise (11)

Define an interpreter:

```
eval :: Expr -> Env Int -> [Int]
```

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿? ? ? ? ? ? ? ? ?