Computer Science 51: Abstraction and Design in Computation

Austin Li
awli@college.harvard.edu

Spring 2022

Abstract

These are notes[1] for Harvard's *Computer Science 51*, an undergraduate class on abstraction and design, as taught by Professor Stephen Chong in Spring 2022.

Course description: Fundamental concepts in the design of computer programs, emphasizing the crucial role of abstraction. The goal of the course is to give students insight into the difference between programming and programming well. To emphasize the differing approaches to expressing programming solutions, you will learn to program in a variety of paradigms – including functional, imperative, and object-oriented. Important ideas from software engineering and models of computation will inform these different views of programming.

These notes cover Professor Stuart M. Shieber's Abstraction and Design in Computation

Contents

0.1 Logistics 6
0.1.1 Course overview 6

1 Introduction **7**
1.1 Extended example 7
1.2 Programming as design 7
1.3 The OCaml programming language 8

2 A Cook's tour of OCaml **9**

3 Expressions and the linguistics of programming languages **10**
3.1 Specifying syntactic structure with rules 10
3.2 Disambiguating ambiguous expressions 10
3.3 Abstract and concrete syntax 11
3.4 Expressing your intentions 11
3.4.1 Commenting 11

4 Values and types **12**
4.1 OCaml expressions have values 12
4.1.1 Integer values and expressions 12

[1]With thanks to Eric K. Zhang for the template.

4.1.2 Floating point values and expressions . . . 12
4.1.3 Character and string values . . . 12
4.1.4 Truth values and expressions . . . 12
4.2 OCaml expressions have types . . . 12
4.2.1 Type expressions and typings . . . 13
4.3 The unit type . . . 13
4.4 Functions are values . . . 13

5 Naming and scope **14**
5.1 Naming . . . 14
5.2 Scope . . . 14
5.3 Global naming and top-level let . . . 14

6 Functions **15**
6.1 Multiple arguments and currying . . . 15
6.2 Defining anonymous functions . . . 15
6.3 Named functions . . . 15
6.3.1 Compact function definitions . . . 15
6.3.2 Providing typings for function arguments and outputs . . . 16
6.4 Defining recursive functions . . . 16
6.5 Unit testing . . . 16

7 Structured data and composite types [TO-DO] **17**
7.1 Tuples . . . 17
7.2 Pattern matching for decomposing data structures . . . 17

8 Higher-order functions and functional programming [TO-DO] **18**

9 Polymorphism and generic programming **19**
9.1 Type inference and type variables . . . 19
9.2 Polymorphic map . . . 19
9.3 Regaining explicit types . . . 19
9.4 The List library . . . 19
9.5 Weak variable types . . . 19

10 Handling anomalous conditions **20**
10.1 A non-solution: error values . . . 20
10.2 Option types . . . 20
10.2.1 Option poisoning . . . 20
10.3 Exceptions . . . 20
10.3.1 Handling exceptions . . . 20

10.3.2 Declaring new exceptions . . . 21
10.4 Options or exceptions . . . 21
10.5 Unit testing with exceptions . . . 21
10.6 Problem set 3: Bignums and RSA encryption . . . 21

11 Algebraic data types **22**
11.1 Built-in composite types as algebraic types . . . 22
11.2 Example: Boolean document search . . . 23
11.3 Example: Dictionaries . . . 23
11.4 Example: Arithmetic expressions as an algebraic type . . . 24
11.5 Problem section: Mini-poker . . . 24
11.6 Problem section: Walking trees . . . 24
11.7 Problem section: Gorn addresses in binary search trees . . . 24
11.8 Problem set 4: Symbolic differentiation . . . 24

12 Abstract data types and modular programming **25**
12.1 Modules . . . 25
12.2 A queue model . . . 25
12.3 Signatures hide extra components . . . 26
12.4 Modules with polymorphic components . . . 26
12.5 Abstract data types and programming for change . . . 26
12.5.1 A string set module . . . 26
12.5.2 A generic set signature . . . 26
12.5.3 A generic set implementation . . . 26
12.6 A dictionary module . . . 27
12.7 Alternative methods for defining signatures and modules . . . 27
12.7.1 Set and dictionary modules . . . 27
12.8 Library modules . . . 27
12.9 Problem section: Image manipulation . . . 27
12.10Problem section: An abstract data type for intervals . . . 27
12.11Problem section: Mobiles . . . 27
12.12Problem set 5: Ordered collections . . . 27

13 Semantics: The substitution model **28**
13.1 Semantics of arithmetic expressions . . . 28
13.2 Semantics of local naming . . . 28
13.3 Defining substitution . . . 29
13.3.1 Handling variable scope . . . 29
13.3.2 Free and bound occurrences of variables . . . 29

13.4 Implementing a substitution semantics 29
13.5 Problem section: Semantics of booleans and conditionals 29
13.6 Semantics of function application 29
13.7 Substitution semantics of recursion 29

14 Efficiency, complexity, and recurrences **30**
14.1 The need for abstract notions of efficiency 30
14.2 Two sorting functions 30
14.3 Empirical efficiency 30
14.4 Big-O notation 30
14.4.1 Informal function notation 30
14.4.2 Useful properties of O 30
14.5 Recurrence equations 30
14.5.1 Solving recurrence by unfolding 30
14.6 Problem section: Complexity of the Luhn check 31
14.7 Problem set 6: The search for intelligent solutions 31

15 Mutable state and imperative programming **32**
15.1 References 32
15.1.1 Reference operator types 32
15.1.2 Boxes and arrows 32
15.1.3 References and pointers 32
15.2 Other primitive mutable data types 32
15.2.1 Mutable record fields 33
15.2.2 Arrays 33
15.2.3 References and mutation 33
15.3 Mutable lists 33
15.4 Imperative queues 33
15.5 Hash tables 33

16 Loops and procedural programming **34**
16.1 Loops require impurity 34
16.2 Recursion versus iteration 34
16.2.1 Tail recursion 34
16.3 Saving space 34
16.3.1 Problem section: Metering allocations 34
16.3.2 Reusing space through mutable data structures 34
16.4 In-place sorting 35

17 Infinite data structures and lazy programming **36**

17.1 Delaying computation 36
17.2 Streams 36
17.2.1 Operations on streams 36
17.3 Lazy computation and thunks 37
17.3.1 The Lazy Module 37
17.4 Application: Approximating π 38
17.5 Problem section: Circuits and boolean streams 38
17.6 A unit testing framework 38
17.7 A brief history of laziness 38
17.8 Problem set 7: Refs, streams, and music 38

18 Extension and object-oriented programming **39**
18.1 Drawing graphical elements 39
18.2 Objects introduced 39
18.3 Object-oriented terminology and syntax 40
18.4 Inheritance 40
18.5 Overriding 40
18.6 Subtyping 40
18.7 Problem section: Object-oriented counters 41
18.8 Problem set 8: Force-directed graph drawing 41
18.9 Problem set 9: Simulating an infectious process 41

19 Semantics: The environment model **42**
19.1 Review of substitution semantics 42
19.2 Environment semantics 42
19.2.1 Dynamic environment semantics 42
19.2.2 Lexical environment semantics 43
19.3 Conditionals and booleans 43
19.4 Recursion 43
19.5 Implementing environment semantics 43
19.6 Semantics of mutable storage 43
19.6.1 Lexical environment semantics of recursion 44

20 Final project: Implementing MiniML **45**
20.1 Overview 45
20.1.1 Grading and collaboration 45
20.2 Implementing a substitution semantics for MiniML 45

0.1 Logistics

Announcements

- For next time: Read chapters 1-4 in the course textbook
- Submit the reading survey, work on problem set 0

0.1.1 Course overview

We can review the syllabus. There are not very many lectures in this course. Lectures are optional and recordings be available on Canvas.

Most of the learning occurs in the labs. Labs are small-group sessions that occur on Tuesdays and Thursdays. Labs will be held in 114 Western Ave Rooms 2111 and 2112.

We can submit labs as often as we want, up until virtual quizzes are released on Sundays. There are nine problem sets, two exams, and one final project.

1 Introduction

Without abstraction, there are only details. And through abstraction — forgetting differences, generalizing — that we can get control of the sheer daunting complexity of controlling a computer.

Definition 1.1 (Abstraction)**.** *Abstraction* is the process of viewing a set of apparently dissimilar things as instantiating an underlying identity.

Example 1.2. We can view a field of a hundred flowers as a set of individual flowers. For a botanist, however, these dissimilar individuals are all instances of a type, the genus Tulipa, the tulips.

Capturing innumerable plants into a hierarchy of abstract families, genera, and species, the complexity of plant life becomes manageable.

This book has two objectives: to introduce the reader to a broad variety of abstraction mechanisms and to apply these abstractions to implement more elegant code.

We can think about *imperative programming* as a derivative of the Turing machine, where we read languages as a procedure to change an internal *state variable*. These languages use state variables and for loops as their basic abstraction methods, and notable examples include C, Python, etc.

The other paradigm of programming, *functional programming*, is focused on functions as parameters.

1.1 Extended example

Example 1.3 (Tiling a bathroom)**.** We can consider the problem of finding the largest square tiling of a 28×20 bathroom floor. More generally, tiling an $a \times b$ bathroom floor. We can consider finding the greatest common divisor by decrementing one from $\min(a, b)$ until we find an integer $d : d|a, d|b$.

There is an explicit way to write the code and a recursive definition.

There is yet a better algorithm devised by Euclid in his *Elements*. The simple observation is that any square tiling of $a \times b, a > b$ must tile $b, a \mod b$. This is known as *Euclid's algorithm* and is much more efficient that the countdown algorithm.

1.2 Programming as design

Euclid's algorithm for greatest common divisor shows us that there is more than one way to solve a problem and some ways are better than others. We can judge the quality of a solution by

- Succinctness
- Efficiency
- Readability
- Maintainability
- Provability
- Testability
- Beauty

All these practices are centered around *design*.

Definition 1.4 (Design)**.** *Design* is the navigation of a space of options, generated by applicable tools, in search of the good, as measured along multiple dimensions. For computer programming, the tools are the abstraction mechanisms provided by a programming language.

Functions are one of these platforms for abstraction mechanisms.

Note. Functions constitute a complete universal computational mechanism. Developed from Alonzo Church's *lambda calculus* — a logical system that included functions and their application — in the absence of data types or structures.

Turing later showed that this is equivalent to a Turing machine. This argument for two universal computational models is known as the *Church-Turing Thesis*.

We concentrate on the following abstraction mechanisms with the associated programming paradigms

Abstraction	Programming paradigm
functions	functional programming
algebraic data types	structure-driven programming
polymorphism	generic programming
abstract data types	modular programming
mutable state	imperative programming
loops	procedural programming
lazy evaluation	programming with infinite data structures
object dispatch	object-oriented programming
concurrency	concurrent programming

Table 1: Some abstraction mechanisms and paradigms

1.3 The OCaml programming language

OCaml is a multi-paradigm programming language that supports functional, imperative, object-oriented, and other mechanisms and paradigms in the above table.

OCaml turns out to be a good pedagogical resource.

2 A Cook's tour of OCaml

In the OCaml interactive prompt, # indicates that we can enter an OCaml expression and ;; indicates the end of the expression. The system reads the expression, evaluates it, and prints the indication of the result, and loops back to provide another prompt. This is why the OCaml interactive system is referred to as the *REPL*.

3 Expressions and the linguistics of programming languages

One truth from linguistics is that expressive units of natural languages or *expressions* have hierarchical structure.

Definition 3.1 (Syntax)**.** Characterizing what the well-formed and structured phrases of a language is *syntax.*

3.1 Specifying syntactic structure with rules

Consider expressions in English. For example, noun phrases like *party, drinker, tea* or putting together a noun phrase and a noun like *tea party*, or putting together an adjective and noun phrase like *iced tea.* We can write the rules as follows:

$$\langle\texttt{nounphrase}\rangle ::= \langle\texttt{noun}\rangle, \tag{1}$$
$$\langle\texttt{nounphrase}\rangle ::= \langle\texttt{adjective}\rangle\langle\texttt{nounphrase}\rangle, \tag{2}$$
$$\langle\texttt{nounphrase}\rangle ::= \langle\texttt{nounphrase}\rangle\langle\texttt{noun}\rangle \tag{3}$$

The notation ::= is read as "can be composed from". The notation for presenting syntax is called *Backus-Naur form* (BNF), named for John Backus and Peter Naur. We can put these together and write

$$\langle\texttt{nounphrase}\rangle ::= \langle\texttt{noun}\rangle|\langle\texttt{adjective}\rangle\langle\texttt{nounphrase}\rangle|\langle\texttt{nounphrase}\rangle\langle\texttt{noun}\rangle \tag{4}$$

Definition 3.2 (Grammar)**.** A specification of a language using rules like this is called a *grammar.*

Definition 3.3 (Ambiguous)**.** We note that a word is *ambiguous* if it can be derived from different syntax analyses.

For example, we have

$$\langle\texttt{nounphrase}\rangle \to \langle\texttt{nounphrase}\rangle\langle\texttt{noun}\rangle \to \langle\texttt{adjective}\rangle\langle\texttt{nounphrase}\rangle\langle\texttt{noun}\rangle \to \texttt{iced tea drinker} \tag{5}$$
$$\langle\texttt{nounphrase}\rangle \to \langle\texttt{adjective}\rangle\langle\texttt{nounphrase}\rangle \to \langle\texttt{adjective}\rangle\langle\texttt{nounphrase}\rangle\langle\texttt{noun}\rangle \to \texttt{iced tea drinker} \tag{6}$$

Definition 3.4 (Semantics)**.** *Semantics* is the characterization o the meanings of expressions on the basis o structure.

We note that all natural languages are ambiguous and often rely on context clues like intonation and others to derive meaning.

3.2 Disambiguating ambiguous expressions

Programming languages have syntactic structure. Consider the BNF rules for simple arithmetic expressions from numbers and binary operations

$$\langle\texttt{expr}\rangle ::= \langle\texttt{expr}_{\texttt{left}}\rangle\langle\texttt{binop}\rangle\langle\texttt{expr}_{\texttt{right}}\rangle|\langle\texttt{number}\rangle, \qquad \langle\texttt{binop}\rangle ::= +|-|\times|\div, \qquad \langle\texttt{number}\rangle ::== 0|1|2|3|\cdots \tag{7}$$

Using these rules, we can build arithmetic expressions for $3 + 4 \times 5$. We do not, however, have intonation or shared context to disambiguate expressions. We instead use the *order of operations* as convention. We refer to this kind of priority in operators as their *precedence*, with higher precedence operators $(\times, \div)$ appearing lower in the tree than lower precedence operators (+,-).

Thus $3 + 4 \times 5 \neq 35, 3 + 4 \times 5 = 23$. For operators of the same precedence, we rely on the *associativity* of an operator.

Definition 3.5 (Associativity)**.** An operator is *left associative* if the operations are applied starting with the left one. For example, $-$. We evaluate $5 - 4 - 1 = (5 - 4) - 1$.

Other operators can be *right associative*, for example the `**` operator in OCaml. We have `2 ** 2 ** 3 = 2 ** (2 ** 3)`.

Associativity and precedence conventions pick out the abstract structure of concrete expressions. We can use annotations to enforce a particular structure, using *parentheses* to override conventional rules. Thus $(3 + 4) \times 5 = 35$.

3.3 Abstract and concrete syntax

The right way to think about expressions is as a hierarchically structured object, often depicted as trees.

Definition 3.6 (Abstract and concrete syntax)**.** We will use *abstract syntax* for expressions qua structured objects and *concrete syntax* for their linear-notated manifestations.

3.4 Expressing your intentions

It is through expressions of a programming language that programmers express intentions to a computer. The computer interprets the expressions to carry out intentions.

In order to remember intentions of code, we will follow the fundamental principle: *make your intentions clear.*

3.4.1 Commenting

The audience for *comments* on code is human readers. They document the intended workings of a program for human readers.

In OCaml, comments are marked by surrounding them with delimiters `(* <> *)`. Comments should describe the *why* rather than the *how* of a program.

There are other aspects that can be freely employed. For example, spaces, newlines, indentations, and variable names can be used to make intentions clear by laying out the code in a way that emphasizes its structure or internal patterns.

4 Values and types

OCaml is a value-based, strongly, statically, implicitly typed, functional programming language. We will discuss these aspects of the language.

4.1 OCaml expressions have values

The OCaml language is a language for calculating *values*. The process of calculating the value is *evaluation*.

4.1.1 Integer values and expressions

The standard arithmetic operators apply. Integer negation is ∼`-`, a tilde and a hyphen.

A full set of built-in operators is available here in the standard library.

4.1.2 Floating point values and expressions

Real numbers are represented using *floating point* approximation. *Floating point literals* can be expressed in decimal notation, exponents, and hexadecimal.

The floating point operators are

$$\texttt{+.},\quad \texttt{-.},\quad \texttt{*.},\quad \texttt{/.} \tag{8}$$

Note. These operators are distinct from the integers.

4.1.3 Character and string values

Text is represented as a strings of *characters*. Character literals are given single quotes, for example `‘a’`, `‘X’`, `‘3’`. Special characters are escaped with a backslash, such as the quote character `"`.

String literals are given in double quotes, for instance `‘‘first"`, `‘‘and second"`. They can be concatenated with the ˆ operator.

4.1.4 Truth values and expressions

There are two *truth values*, indicated in OCaml by literals true and false. The truth values can be operated on with logical operators `&&,` ||, `not` for $\wedge, \vee, \neg$.

The equality operator tests two values for equality and returns a boolean. There are other *comparison operators* as well, like `<, >, <=, >=, <>` where `<>` is not equal.

The OCaml *conditional* expression follows the template

$$\text{if } \langle\texttt{expr}_{\texttt{test}}\rangle \texttt{ then } \langle\texttt{expr}_{\texttt{true}}\rangle \texttt{ else } \langle\texttt{expr}_{\texttt{false}}\rangle \tag{9}$$

where it returns $\langle\texttt{expr}_{\texttt{true}}\rangle$ if the test expression is true and $\langle\texttt{expr}_{\texttt{false}}\rangle$ if the test expression is false.

4.2 OCaml expressions have types

Every expression of the language is associated with a type, making OCaml a *typed* language.

OCaml is *statically typed*, in that the type of an expression can be determined by examining the expression in its context. It is not necessary to run the code in which an expression occurs to determine the type of an expression, as in a *dynamically typed* language like Python or Javascript.

Moreover, OCaml is *strongly typed*, values may not be used in ways inappropriate for their type. One consequence is that functions only apply to values of certain types and only return values of certain types.

Because of this, the system can tell the programmer when type constraints are violated *before the program is run*, preventing bugs before they happen.

4.2.1 Type expressions and typings

Every type has a name and these names are given as *type expressions.* Each *atomic type* has its own name.

Some types include integers, floating point numbers, characters, strings, truth values, and units.

We can test types using the : operator, sometimes read as “the". This is called *typing*, and an incorrect typing will cause the REPL to generate an error. (ie. `(42 : float)`)

OCaml expressions are *implicitly typed.* Although all expressions have types, and types of expressions can be annotated using types, the programmer doesn’t need to specify those types in general. The OCaml interpreter can deduce the types of expressions at compile time using a process called *type inference.*

4.3 The unit type

`()` is the only value of the *unit* type. This will be more relevant in future chapters.

4.4 Functions are values

Functions in OCaml are *first-class values*, they can be passed as arguments to functions or returned as the value of functions. Functions that take functions as arguments are referred to as *higher-order functions* and the powerful programming paradigm that makes full use of this capability is *higher-order functional programming.*

A functions’ type expression is formed by placing the symbol `->` (read “arrow” or “to”) between the argument type and the output type. For example, the type of the `sqrt` function is `float -> float`, read "float arrow float" or "float to float".

5 Naming and scope

5.1 Naming

Variables can be thought of as names for values. To introduce a variable, use the local naming construct:

$$\texttt{let } \langle\texttt{var}\rangle : \langle\texttt{type}\rangle = \langle\texttt{expr}_{\texttt{def}}\rangle \texttt{ in } \langle\texttt{expr}_{\texttt{body}}\rangle \tag{10}$$

$\langle\texttt{var}\rangle$ is a variable, or name of a value of the given $\langle\texttt{type}\rangle$, $\langle\texttt{expr}_{\texttt{def}}\rangle$ is an expression defining a value, and $\langle\texttt{expr}_{\texttt{body}}\rangle$ is an expression within which the variable can be used as the name for the defined value. We say that the expression *binds* the name $\langle\texttt{var}\rangle$ to the expression $\langle\texttt{expr}_{\texttt{def}}\rangle$ for use in $\langle\texttt{expr}_{\texttt{body}}\rangle$. The let construction thus a *binding construct.*

Example 5.1 (Let construct)**.** Consider

$$\text{let pi : float} = 3.1415 \text{ in} \tag{11}$$

Let expressions can be used as first class values, so they may be embedded in other let expressions to get the effect of defining multiple names.

Note. The $\langle\textit{type}\rangle$ can be omitted and OCaml can infer types, which is why we say OCaml is implicitly typed.

5.2 Scope

Names defined in a let expression are *local* to the expression and unavailable outside of the body of the expression.

Definition 5.2 (Scope)**.** We say that the *scope* of the variable — that is, the code within which the variable is available as a name of the defined value — is the body of the let expression.

Note. The scope of a local let naming does not include the definition itself (ie. `let x = x + 1 in` $\langle\texttt{expr}_{\texttt{body}}\rangle$ is not well defined).

The rule used in OCaml (and most modern languages) is that the occurrences are bound by the *nearest enclosing binding construct for the variable.* Thus, when an inner binder for a variable falls within the scope of an outer binder for the same variable, the outer variable is inaccessible in the inner scope. In this case, we say that the outer variable is *shadowed* by the inner variable.

5.3 Global naming and top-level let

OCaml provides a global naming construct as well. By leaving off the 'in $\langle\texttt{expr}_{\texttt{body}}\rangle$' part of the let construct, the name can continue to be used thereafter; the scope of the naming extends through the remainder of the REPL session or to the end of the program file.

Note. This is distinct from assignment in imperative programming languages. While it may look like we are assigning values to a variable, actually, we are creating new names for values.

Global naming is available only at the top level. A global name cannot be defined from within another expression, for instance, the body of a local let.

6 Functions

Definition 6.1 (Function)**.** A *function* is a mapping from an input — the *argument* — to an output — the function's *value.*

We make use of a function by *applying* it to its argument.

In Church's *lambda calculus*, we simply prefix the function to its argument and instead use parentheses for grouping. This is true in OCaml. The function merely precedes its argument without parentheses. For example, instead of `f(1)`, the notation is `f 1`. Recall from Section 4.4 that functions (as all values) have types, which can be expressed as type expressions using the `->` operator. For instance, the successor function has the type given by the type expression `int -> int` and the "evenness" function the type `int -> bool`.

6.1 Multiple arguments and currying

Prefix notation is only appropriate if functions take one argument. For multiple arguments, we can think about a function `f(1,2,3)`. We can think about `f` as taking one argument, returning a function that takes the second argument, and returning a function that takes the final argument.

The function takes the three arguments *one at a time.* The trick was discussed by Schönfinkel (1924) and is referred to as a *currying* function.

OCaml makes extensive use of currying, and language constructs facilitate its use. For example, the `->` type expression is right associative.

6.2 Defining anonymous functions

Definition 6.2 (Anomymous function)**.** An *anonymous function* is a function without a name.

We can construct an anonymous function with the construct

$$\texttt{fun}\ \langle\texttt{var}\rangle : \langle\texttt{type}\rangle\ \texttt{->}\ \langle\texttt{expr}\rangle \tag{12}$$

⟨`var`⟩ is a variable name with a given ⟨`type`⟩ and output ⟨`expr`⟩. The fun construct is a binding construct. It binds occurrences of a variable in its scope, which is the body of the `fun`, the expression ⟨`expr`⟩ after the arrow.

Example 6.3 (Anonymous functions)**.** Consider

```
fun x -> 2 * x ;;                                  (13)
```

6.3 Named functions

We can name functions with let.

Example 6.4 (Named functions)**.** Consider

```
let double = fun x -> 2 * x ;;                     (14)
```

6.3.1 Compact function definitions

OCaml provides a simpler syntax. OCaml has a similar phrasing

$$\texttt{let}\ \langle\texttt{var}_{\texttt{func}}\rangle\langle\texttt{var}_{\texttt{arg}}\rangle = \langle\texttt{expr}\rangle \tag{15}$$

This compact syntax for function definition is an example of *syntactic sugar.*

Example 6.5 (Compact definition)**.** Consider the syntactic sugar:

```
let double x = 2 * x in
```
(16)

6.3.2 Providing typings for function arguments and outputs

We can provide typing for the variable being defined, like

```
let hypotenuse :  float -> float -> float = fun x -> fun y -> sqrt (x ** 2.  +.  y ** 2.)  ;;
```
(17)

and it is good practice to do for top-level definitions.

6.4 Defining recursive functions

We can consider the *factorial* function. For a recursive definition, we must add the `rec` keyword after let.

The rec keyword means that the scope of the let includes not only the body but also its definition. The code is

```
let rec fact (n :  int) :  int = if n = 0 then 1 else n * fact (n - 1)
```
(18)

6.5 Unit testing

We can assure that our code is correct through *formal verification* of software — proving that the code does what we want it to do.

If we can't have a proof, we need to test that it generates the appropriate values on a full range of test cases. This is an approach called *unit testing.*

7 Structured data and composite types [TO-DO]

7.1 Tuples

Definition 7.1 (Tuple)**.** A *tuple* is a fixed length sequence of elements.

The *value constructor* for tuples is an infix comma. For example, `3, true`.

The type of a pair is determined by the types of its parts. For example, we giving the types of the parts combined using the infix *type constructor* `*`. For instance, the pair `3, true` is of type `int * bool` (read, "int cross bool").

7.2 Pattern matching for decomposing data structures

The match construction is used to perform this matching and decomposition. The general form of a match is

$$\texttt{match <expr> with| <pattern}_1\texttt{> -> <expr}_1\texttt{> | <pattern}_2\texttt{> -> <expr}_2\texttt{>} \tag{19}$$

Definition 7.2 (Anonymous variable)**.** An *anonymous variable* is a a variable starting with the underscore character. This codifies the programmer's intention that the variable not be used, and disables the warning message.

8 Higher-order functions and functional programming [TO-DO]

9 Polymorphism and generic programming

Recall the map function from previous chapters. We can implement a map function for lists *generically*, while still obeying the constraint that whatever type the list elements are, they are appropriate to apply the function to.

9.1 Type inference and type variables

One solution is type inference. We can infer the type of an input and output from the definition, using operators.

We can consider the identity function:

```
let id x = x ;;
```
(20)

Because `x` is not involved in any applications of the definition of `id`, it ha no type constraints.

Definition 9.1 (Polymorphic functions and polymorphic types)**.** A function is *polymorphic* if it doesn't have a fully instantiated type.

To express polymorphic types, we need to extend the type expression language. We can use *type variables* as identifiers with a prefix quote, e.g. `'a, 'b, 'c` and so forth, reading them as their corresponding Greek letter.

9.2 Polymorphic map

We can remove the typings in the definition of map to create a polymorphic version of the function.

9.3 Regaining explicit types

We can also make the types explicit again in function definitions.

9.4 The List library

OCaml, like Python, comes with a large set of libraries. The `List` library contains a lot of useful abstractions including `map, fold, filter`, among others. The full documentation for the List module is provided.

9.5 Weak variable types

The `List` module provides polymorphic `hd` and `tl` functions for extracting the head and tail of a list.

Definition 9.2 (Weak type variables)**.** *Weak type variables*are variables that maintain their polymorphism only temporarily, until the first time they are applied.

When a function with these weak type variables is applied to arguments with a specific type, the polymorphism of the function disappears.

10 Handling anomalous conditions

On occasion, an *anomaly* occurs that a function can't handle. The function can return a value that indicates the anomaly. We will discuss how to handle these errors and anomalies.

Example 10.1 (Median values)**.** We can consider a function that calculates the median number in a list of integers.

There is one anomalous case: what is the median of an empty list?

10.1 A non-solution: error values

We can consider returning special *error values.* There are problems: it can lead to type instantiation (replacing generalized types) and manifests in-band signaling.

In-band signaling is using valid values to indicate errors. This could raise problems.

10.2 Option types

The function can return an out-of-band `None` value. We can use the ostfix type constructor `option` to create an option type. There are two value constructors: `None` and `Some`, which denotes an anomalous value and a value of the base type, respectively.

10.2.1 Option poisoning

There is a problem with using options to handle anomalies. To extract values from the output of one of these functions, we need to extract values by passing on `None`s.

Definition 10.2 (Option poisoning)**.** *Option poisoning* is the phenomenon that occurs when we introduce option types. We lose the elegance of functional programming — the ability to embed function applications with other functional applications.

10.3 Exceptions

Another solution is to raise an *exception.* In this case, the execution of the function simply *stops* — the function cannot return a value, which is appropriate because there is no appropriate value to return.

Example 10.3. We can consider a version of `nth` that raises an exception.

```
let rec nth (lst :  'a list) (n :  int) :  'a = match lst with | [] -> raise Exit
```
(21)

Functions that utilize this as a way to indicate errors keeps the `'a` type, not `'a option`.

10.3.1 Handling exceptions

The exception will propagate to the larger function if the error is raised in a subroutine.

If we do **not** want the exception to propagate to the top level, we can handle the exception ourselves with the following construct:

$$\texttt{try}\ \langle\texttt{expr}\rangle\ \texttt{with}\ \langle\texttt{match}\rangle \tag{22}$$

where ⟨`expr`⟩ is an expression that may raise an exception and ⟨`match`⟩ is a pattern match with one or more branches.

Example 10.4 (Zipping lists)**.** We can consider a zipping list. there are two cases that are unmatched in some normal implementation, when one of the lists is empty. We simply raise an error.

10.3.2 Declaring new exceptions

Exceptions are first-class values of type `exn`. There are multiple value constructors: `Exit`, `Failure`, `Invalid_argument`. We can also define new constructors like `Timeout` or `UnboundVariable of string`.

10.4 Options or exceptions

We can use *either* options or exceptions. This is a design decision and there is no universal correct answer.

Options are explicit and exceptions are implicit. Options indicate that anomalies might occur and handle them. Exceptions are more concise, it doesn't impinge on the data and does not poison any downstream use of the data.

We can consider the rarity of errors in deciding between options or exceptions.

10.5 Unit testing with exceptions

We can use unit tests to test for exceptions. Examples are provided in this section of the textbook.

10.6 Problem set 3: Bignums and RSA encryption

This section provides background for the third problem set. Read before starting!

11 Algebraic data types

Data types can be divided into *atomic* types with atomic constructors and *composite* types with parameterized type constructors like `<> * <>`, `<> list` and `<> option`.

All of the built-in composite types allow building data structures by combining two methods:

1. **Conjunction.** Multiple components can be conjoined to form a composite value containing *all* components

   Example 11.1. For example, values of type `int * float` are formed as a conjunction of two components.

2. **Alternation.** Multiple components can be disjoined, serving as alternatives to form a composite value containing *one* of the values.

   Example 11.2. Values of type `int list` are formed by an alternation of two components.

Definition 11.3 (Algebraic data types)**.** Data types built by conjunction and disjunction are called *algebraic data types.*

These algebraic types can be a foundational construct of the language.

OCaml inherits from its heredity the ability to define new algebraic data types as user code.

Example 11.4 (DNA bases)**.** We can define an algebraic data type for DNA called `base` as follows:

```
type base = G |C | A | T ;;                    (23)
```

where we use vertical bars as separators.

Definition 11.5 (Variant type)**.** This kind of type declaration defines a *variant type*, which lists a set of alternatives.

We can then refer to values of that type once we declare the `base` type. We can use pattern matching on these types as well.

We can then define the `dna` type. These values can be `Nil`, which indicates en empty sequence, and the `Cons` constructor to take two arguments (uncurried so a single pair argument). We see this below:

Example 11.6 (DNA)**.** Consider the following type definition for DNA:

```
type dna = | Nil | Cons of (base * dna) ;;                    (24)
```

The `Cons` constructor is a pair and conjoins a `base` element to another `dna` sequence.

This type is defined recursively. Recursion is useful to define data types of arbitrary size.

11.1 Built-in composite types as algebraic types

The `dna` type looks like the `list` type built into OCaml but with `base` elements.

Example 11.7 (List definition)**.** Consider the following definition for a list:

```
type 'a list = Nil | Cons of 'a * 'a list ;;                    (25)
```

We use these examples not to *recreate* these data types, violating the edict of redundancy, but to demonstrate the power of algebraic data type definitions.

11.2 Example: Boolean document search

This is not the first time we've seen algebraic data types, recall the `keyword` type from section 7.4.

We also recall `record` types. We can consider an application of this in the `document` type:

```
type document = { title : string; words : string list } ;;        (26)
```

We can also implement a `document list`. We may try to query documents with particular patterns of words. We can do this using *boolean queries.* This allows for different query types. We can instantiate the idea in a variant type definition:

```
type query = | Word of string | And of query * query | Or of query * query ;;        (27)
```

We can evaluate queries against a document by writing a function:

```
let rec eval ({title; words} : document) (q : query) : bool =
                match q with
                | Word word -> List.mem word words
                | And (q1, q2) -> ...
                | Or (q1, q2) -> ...  ;;
```

Note. See the full implementation in textbook p.155. Also it is important to familiarize ourselves with the `List` library.

We can simplify redundancies using the pattern construct

$$\langle\texttt{pattern}\rangle\ \texttt{as}\ \langle\texttt{variable}\rangle. \tag{28}$$

This kind of pattern pattern matches against the ⟨`pattern`⟩ as well as binding the ⟨`variable`⟩ to the expression being matched.

We consider rewriting

```
let rec eval ({words; _} as doc : document) (q : query) : bool = ...        (29)
```

Note. The complete function is specified on pages 157-158.

Note. OCaml also supports a *bckwards application* infix operator `|>` to help make code more readable.

Example 11.8 (Succ operator)**.** The following code does the same thing:

```
succ 3 ;;    3 |> succ ;;        (30)
```

and returns `4`. We note that `succ` increments an integer.

11.3 Example: Dictionaries

A dictionary is a data structure that manifests a relationship between a set of *keys* and *values.* We can define a polymorphic dictionary given by

```
type ('key, 'value) dictionary = { keys : 'key list; values : 'value list } ;;        (31)
```

This definition of dictionaries allow keys without values. This will complicate `lookup` functions.

We then arrive at the *edict of prevention: make the illegal inexpressible.*

This edict challenges us to find an alternative structure where this mismatch between keys and values cannot occur. Consider this definition of dictionaries as a list of pairs of keys and values:

```
type ('key, 'value) dict_entry = { key :  'key; value :  'value }
                   and ('key, value) dictionary = ('key, 'value) dict_entry list ;;  (32)
```

This guarantees that every dictionary is a list whose elements each have a key and a value. We cannot have unequal keys and values.

11.4 Example: Arithmetic expressions as an algebraic type

We can use algebraic data types to capture languages. The language of simple integer arithmetic expressions is defined by a grammar. We can express this in Backus-Naur form by

$$\langle\texttt{expr}\rangle \texttt{ ::= } \langle\texttt{integer}\rangle \texttt{ | } \langle\texttt{expr}_1\rangle \texttt{ + } \langle\texttt{expr}_2\rangle \texttt{ | } \langle\texttt{expr}_1\rangle \texttt{ - } \langle\texttt{expr}_2\rangle \texttt{ | } \ldots \tag{33}$$

This is the abstract syntax of the language. We can leave precedence, associativity of operators, and parentheses implicit.

The definition of the grammar is given trivially:

```
type expr = Int of int | Plus of expr * expr | Minus of expr * expr
                        | Times of expr * expr | Div of expr * expr | Neg of expr ;;  (34)
```

We can define an evaluate function to evaluate the expressions.

11.5 Problem section: Mini-poker

11.6 Problem section: Walking trees

11.7 Problem section: Gorn addresses in binary search trees

11.8 Problem set 4: Symbolic differentiation

This section gives background for problem set four.

12 Abstract data types and modular programming

We can consider a fundamental data structure, the *queue*.

Definition 12.1 (Queue)**.** A *queue* is a collection of elements that admits of operations like creating an empty queue, adding elements one by one, and removing them one by one, called *enqueueing* and *dequeuing*, respectively. The common term for this regimen is *first-in-first-out*, or FIFO.

We can define the queue data type using the list data type.

The full implementation is on page 177 and 178.

Note. We want the data structures to not support invalid operations, like reversing a queue. We then want to enforce restraints on the operations applicable to a data structure to preserve invariants.

The key idea to enforce invariants is to provide an *abstract data type*, a data type definition that provides a concrete implementation *and* enforces operations that can be performed. The allowed operations are specified in a *signature*.

The signature specifies an interface to using the data structure, which serves as an *abstraction barrier* — that is, only the aspects of the implementation specified on the signature may be made of use.

We then arrive at the *edict of compartmentalization*: limit information to those with a need to know.

Example 12.2. Following the edict of compartmentalization, all a user needs to know about an implementation is the types of operations involving queues, namely enqueue and dequeue.

12.1 Modules

In OCaml, abstract data types are implemented using *modules*. A module is specified by placing definitions of components between the keywords `struct` and `end`:

$$\texttt{struct}\ \langle\texttt{definition}_1\rangle\ \langle\texttt{definition}_2\rangle\ \texttt{..}\ \texttt{end} \tag{35}$$

where each ⟨`definition`⟩ is a definition of a type or value. Modules are named using the `module` construct:

$$\texttt{module}\ \langle\texttt{module name}\rangle = \langle\texttt{module definition}\rangle \tag{36}$$

12.2 A queue model

We can see the queue module on page 181.

Note. Even though we can define this module, we note that *nothing restricts* us from using arbitrary aspects of implementation, like *reversing* the queue.

We can use signatures to restrict the use of components of a module, like a type restricts the use of a value.

This restricts the types of operations that we can do.

The notation for specifying signatures is similar to modules, except we use the `sig` construct:

```
module type ⟨module type⟩ = sig
        ⟨definition₁⟩ ⟨definition₂⟩ ⟨definition₃⟩ .  .  .
    end
```

where all modules of type ⟨`module type`⟩ is constrained by the definitions in the signature.

Note. Whereas the module implementation uses the `let` construct, the signature uses the `val` construct, which provides a name and a type, but no definition.

We can extend the analogy between signatures and types further by specifying that a module satisfies and is constrained by a signature with notation:

$$\texttt{module}\ \langle\texttt{module name}\rangle\ \texttt{:}\ \ \langle\texttt{signature}\rangle = \langle\texttt{module definition}\rangle \tag{37}$$

12.3 Signatures hide extra components

If a module defines more components than its signature, any function or value in the module that is not specified in the signature (and any other functions that depend on these functions or values) cannot be used.

In general, only the aspects of a module consistent with its signature are visible outside of its implementation to users of the module. All other aspects are hidden behind the abstraction barrier.

A fundamental role of modules and their signatures is to establish these abstraction barriers so that information about how data types happen to be implemented can't leak out and be taken advantage of.

Examples can be seen on page 186-187 in the `ORDERED_TYPE`.

12.4 Modules with polymorphic components

We can use polymorphic types in our module and signature definitions.

12.5 Abstract data types and programming for change

One of the primary advantages of using abstract data types is that by hiding the data type implementations, the implementations can be changed without affecting users of the data types.

Example 12.3. We can see an implementation of `eval` for queries using the `Hashtbl` module.

12.5.1 A string set module

This section provides an implementation of string sets.

12.5.2 A generic set signature

We consider a case where the abstraction barrier is too strict. There are cases where we want the user of the module to have access to the implementation of the `element` type.

We can define *slightly* less abstract signatures using *sharing constraints*, which argument a signature with one or more type equalities.

12.5.3 A generic set implementation

We can implement different types of sets by changing the element type.

We can package some types and related values and functions, transforming one module to another module using *functors*.

Definition 12.4 (Functor)**.** A *functor* is a function that maps modules to modules.

We can consider a functor that takes a module with the `ORDERED_TYPE` signature and delivers a `SET` implementation:

```
module MakeOrderedSet (Elements :  ORDERED_TYPE) : (SET with type element = Elements.t)
```
(38)

12.6 A dictionary module

Here, we implement a dictionary.

12.7 Alternative methods for defining signatures and modules

We have seen two ways to define a signature explicitly.

The first is to name the signature using `module type` and use the name in defining the module

```
module type SIG_NAME = sig .  .  .  end ;;
```
(39)

```
module ModuleName :  SIG_NAME = struct .  .  .  end ;;
```
(40)

and the second to place an unnamed signaure directly constraining the module definition:

```
module ModuleName :  sig .  .  .  end = struct .  .  .  end ;;
```
(41)

the third way is used within OCaml's own implementation of library modules. All components in a `.ml` file *automatically* constitute a module, generated by converting the first letter of the filename to uppercase.

12.7.1 Set and dictionary modules

A file for generating set modules can be packaged into a single module.

We see an implementation of this on the following few pages.

12.8 Library modules

OCaml provides many implementations of these modules discussed in the chapter in *library modules.*

12.9 Problem section: Image manipulation

12.10 Problem section: An abstract data type for intervals

12.11 Problem section: Mobiles

12.12 Problem set 5: Ordered collections

13 Semantics: The substitution model

Semantics is about what expressions *mean.* The *formal, rigorous, precise* semantics of a programming language is useful.

There are three reasons that formalizing a semantics with mathematical rigor is beneficial: mental hygiene, interpreters, and metaprogramming.

In this chapter, we introduce a technique for giving a semantics to small subsets of OCaml. The method of providing formal semantics we introduce is called *large-step operational semantics*, based on the *natural semantics* of scientist Gilles Kahn.

The semantics is *formal* and *operational* because it relies on manipulations on the forms of notations and we specify what programs *evaluate to.*

13.1 Semantics of arithmetic expressions

Recall the semantics of arithmetic in BNF and tree form. We write $P \Downarrow v$ to mean expression P evaluates to value v.

Definition 13.1 (Values)**.** The *values* are the results of evaluation.

Then we have

$$\bar{n} \Downarrow \bar{n} \tag{42}$$

where we use n to stand for any integer and the bar notation to denote the OCaml numerical encoding of the number n.

Example 13.2 (Addition)**.** Using this notation, we can write

$$\begin{aligned} P + Q \Downarrow& \\ &|P \Downarrow \bar{m} \\ &|Q \Downarrow \bar{n} \\ &\Downarrow \overline{m + n} \end{aligned}$$

We can define similar rules for division and multiplication.

13.2 Semantics of local naming

The ⟨`expr`⟩ language defined in the grammar includes a local naming construct, expressed with `let <> in <>`.

We will take the meaning of the local name construct to work by *substituting the value of the definition for occurrences of the variable in the body.*

We use the following evaluation rule:

$$\texttt{let } x = D \texttt{ in } B \Downarrow \tag{43}$$
$$|D \Downarrow v_D \tag{44}$$
$$|B[x \mapsto v_D] \Downarrow v_B \tag{45}$$
$$\Downarrow v_B \tag{46}$$

where $Q[x \mapsto P]$ represents substituting P for occurrences of x in Q.

Example 13.3 (Multiplication)**.** We have

$$\texttt{(x * x)[x} \mapsto \texttt{5] = 5 * 5} \tag{47}$$

13.3 Defining substitution

Because substitution is central in the semantics of the language, this approach to semantics is referred to as *substitution semantics.*

13.3.1 Handling variable scope

We need to take care of variable scope.

13.3.2 Free and bound occurrences of variables

Definition 13.4 (Bound variables)**.** A binding construct like `let` or `fun` *binds* the variable that it introduces. The variable occurrence is said to be *bound.* A variable occurrence is *free* if it is not bound.

Example 13.5. Consider

```
fun x -> x + y
```
(48)

Here, `x` is bound and `y` is free.

13.4 Implementing a substitution semantics

Here, we implement the `eval` function.

13.5 Problem section: Semantics of booleans and conditionals

Exercises for practice.

13.6 Semantics of function application

We can introduce anonymous functions and their application. The implementation is on page 249, but the derivations can be a bit hairy.

13.7 Substitution semantics of recursion

Occurrences of the name *definiendum* in the body are properly replaced with the *definiens*, but occurences in the definiens itself are not.

14 Efficiency, complexity, and recurrences

Definition 14.1 (Efficiency)**.** We say that some agent is *efficient* if it makes the best use of a scarce resource to generate a desired output.

We are often concerned about the efficiency of our programs, but beware of *premature optimization.*

14.1 The need for abstract notions of efficiency

Efficiency and complexity is dependent on input and computation. We will often consider the *worst-case complexity* of the algorithms we write.

14.2 Two sorting functions

We can consider *insertion sort* by recursively placing the element in a list in its appropriate position.

There is a *merge sort* algorithm that merges two lists by recursively sorting the halves.

14.3 Empirical efficiency

The recurrence relations are given by

$$T_{is}(n) = a \cdot n^2 + b, \qquad T_{ms}(n) = c \cdot n \log n + d \tag{49}$$

14.4 Big-O notation

We often talk about algorithms using big-O notation, which characterizes *asymptotic* behavior. We say that $g(n) \in O(f(n)) \implies \exists c \in \mathbb{N} : g(n) \leq c \cdot f(n)$.

14.4.1 Informal function notation

We often just use n as our variable for complexity.

14.4.2 Useful properties of O

We have the following:

- $f \in O(f)$
- $g \in O(f) \implies g + k \in O(f), k \cdot g \in O(f)$
- $f \in O(n^k), g \in O(n^c), k > c \implies f + g \in O(n^k)$.

14.5 Recurrence equations

We can construct *recurrence relations* to talk about the number of computations required to complete a task.

14.5.1 Solving recurrence by unfolding

There is often no closed-form solution for recurrence relations, but we can *unfold* them to obtain some closed-form solution.

Note (Master theorem)**.** Though we don't cover this in the course, there are theorems that give us closed-form solutions from recurrence relations.

14.6 Problem section: Complexity of the Luhn check

14.7 Problem set 6: The search for intelligent solutions

This is the problem set specification for the sixth homework.

15 Mutable state and imperative programming

The range of programming abstractions presented so far are *pure*, where computation is identified with the evaluation of expressions.

Pure programs have *values* rather than *effects*. The term *side effect* is used for effects that impure programs manifest while being evaluated.

In this chapter, we introduce *imperative programming*, a programming paradigm based on side effects and state change. We will begin with mutable data structures and move onto imperative control structures.

15.1 References

OCaml has *reference types*, kind of like pointers. The constructor `ref` is used to construct reference types, for example `int ref`, `(bool -> int) ref` etc.

We can create a reference to a block of memory storing the integer value `42` with the following:

$$\texttt{let r : int ref = ref 43 ;;} \tag{50}$$

We note that `r` is an immutable name but it is a name for a block of memory that is mutable. We can *dereference* and *update* the stored value. Consider

$$\texttt{!r}, \quad \texttt{r := 21} \tag{51}$$

Here, we dereference `r` with `!r` and we update with `:=`.

15.1.1 Reference operator types

We note that the dereference operator has type

$$\texttt{(!) : 'a ref -> 'a} \tag{52}$$

15.1.2 Boxes and arrows

We can visualize references using *box and arrow diagrams.*

The book provides some examples. We can define notions of equality.

Definition 15.1 (Structural equality)**.** When two values have the same structure, regardless of where they are stored in memory, these two values have *structural equality.*

Definition 15.2 (Physical equality)**.** *Physical equality* holds when two values are the identical physical block of memory.

15.1.3 References and pointers

There is some mapping between *pointers* and references. We note that differences are between memory leaks in C. Memory is automatically reclaimed by the system by a process called *garbage collection.*

15.2 Other primitive mutable data types

There are two other primitive data types: mutable record fields and arrays.

15.2.1 Mutable record fields

Recall that records are mutable. We change mutable records using `<-` instead of `:=`.

15.2.2 Arrays

Arrays can have an arbitrary number of elements all of the same type. Note that we can have

```
let a = Array.init 5 (fun n -> n * n) ;;
```
(53)

to initialize an array and we can change an entry via

```
a.(3) <- 0 ;;
```
(54)

15.2.3 References and mutation

We introduce the *binary sequencing operator* `;` where we write `P; Q` means that we evaluate `P` (and does not return its value) and then evaluates `Q` and returns the value.

We then consider a `bump` function.

Note. We need to avoid the use of a global variable to avoid misuse.

15.3 Mutable lists

A mutable list allows the tail of the list to be updated:

```
type 'a mlist = | Nil | Cons of 'a * ('a mlist ref) ;;
```
(55)

15.4 Imperative queues

Recall the functional queue data structure from chapter 12. We then go through implementations using list references, two stacks, and mutable lists.

15.5 Hash tables

A hash table is a mutable dictionary. Implementation is on page 313. Note that key-value pairs are stored in a mutable array of a given size at an index given by the *hash function.* We then discuss *linear probing* to look at sequential locations in the event of a *hash collision.*

16 Loops and procedural programming

Recall our `length` function from chapter 7. This was a recursive approach.

Another way to think about calculating length is to think about what one *does* to calculate the length. This paradigm is called *procedural programming*, which emphasizes the steps in the procedure to be carried out.

One of the main benefits of the paradigm is *space efficiency.* OCaml supports procedural programming. There are `while` loops:

$$\texttt{while}\ \langle \texttt{expr}_{\texttt{condition}} \rangle\ \texttt{do}\ \langle \texttt{expr}_{\texttt{body}} \rangle\ \texttt{done} \tag{56}$$

There are also `for` loops, where we can count up:

$$\texttt{for}\ \langle \texttt{variable} \rangle = \langle \texttt{expr}_{\texttt{start}} \rangle\ \texttt{to}\ \langle \texttt{expr}_{\texttt{end}} \rangle\ \texttt{do}\ \langle \texttt{expr}_{\texttt{body}} \rangle\ \texttt{done} \tag{57}$$

or count down:

$$\texttt{for}\ \langle \texttt{variable} \rangle = \langle \texttt{expr}_{\texttt{start}} \rangle\ \texttt{downto}\ \langle \texttt{expr}_{\texttt{end}} \rangle\ \texttt{do}\ \langle \texttt{expr}_{\texttt{body}} \rangle\ \texttt{done} \tag{58}$$

16.1 Loops require impurity

In a pure language, expressions always have the same value. Because the counter or iterator will change, this is impure.

16.2 Recursion versus iteration

We can think of recursion as a *stack frame* where we continually add elements to a stack of suspended calls and we evaluate when we reach `length []`.

16.2.1 Tail recursion

Using tail recursion, another implementation of `length` uses the result of the computation in the next iteration and **does not require** storage for suspended computation. We do not need a stack frame.

Definition 16.1 (Tail recursion)**.** A program is *tail-recursive* if the recursive invocation is the result of the invoking call.

Note. `fold_left` is a tail-recursive call and `fold_right` is not tail-recursive.

16.3 Saving space

Procedural programming also allows us to avoid building new data structures.

16.3.1 Problem section: Metering allocations

We can determine how many allocations are going on by metering them. We introduce a module `Metered` and are asked to implement the modules.

16.3.2 Reusing space through mutable data structures

By using imperative techniques, we gain access to incremented values without incurring the cost of further storage.

16.4 In-place sorting

An example of reducing storage requirements is by considering *quicksort*. Quicksort works by selecting a pivot value. The two lists are recursively sorted and concetenated to form a final sorted list.

An implementation of this on pages 326-327.

17 Infinite data structures and lazy programming

Combining functions as first-class values, algebraic data types, and references enables programming with infinite data structures. We will build infinite lists (streams) and infinite trees.

17.1 Delaying computation

OCaml is an *eager* language. We evaluate variables and the body of a function before evaluating a function.

We can consider

```
let rec forever n = 1 + forever n ;;                    (59)
```

and note that this never completes evaluation. This indicates the potential utility of *lazy evaluation*, the ability to *delay* computation until it is needed, at which the computation can be *forced* to occur.

Example 17.1. Conditional expressions delay evaluation until the condition is true. Moreover, a function of a body is not evaluated until the function is applied.

Note. Some languages embraced lazy evaluation as a default, starting with Rod Burstall's Hope language and finding its use in the Haskell language.

We will find lazy evaluation in the creation and manipulation of infinite data structures.

17.2 Streams

There is a new algebraic data type definition for *stream*:

```
type 'a stream = Cons of 'a * 'a stream ;;              (60)
```

We can consider making a stream of ones as follows:

```
let rec ones = Cons (1, ones) ;;                        (61)
```

and we can use operations on the stream `ones`.

Note. This works because the components of an algebraic data type has pointers to their values. We are actually assigning the tail to point at the head, creating a cycle. We note that `<cycle>` appears from the REPL.

17.2.1 Operations on streams

We can consider a definition for `map` on streams.

```
let rec smap (f :  'a -> 'b) (s:  'a stream) :  ('b stream) =
                     match s with | Cons (hd, tl) -> Cons (f hd, smap f tl) ;;  (62)
```

When we run this however, we are blowing the stack because we want to apply `f` to each element in an infinite sequence of ones.

When calculating the result of the map, we need to generate and cons together the head of the list and the tail of the list but the tail involves a `smap`.

We can consider functions as values and achieve delay of computation by taking a stream as a delayed cons, a function from `unit` to the cons. We can write a new definition for streams:

```
type 'a stream_internal = Cons of 'a * 'a stream
                                  and 'a stream = unit -> 'a stream_internal ;;  (63)
```

Then an infinite stream of ones is now defined by

```
let rec ones :  int stream = fun () -> Cons (1, ones) ;;                (64)
```

We can redefine `head` and `tail` functions to return lazy streams. We can redefine `smap` similarly given on page 338.

17.3 Lazy computation and thunks

ecall the definition of streams:

```
type 'a stream_internal = Cons of 'a * 'a stream and 'a stream = unit -> 'a stream_internal ;;
                                                                                         (65)
```

Every time we want to access the head or tail of the stream, we need to rerun the function. We should be able to avoid recomputation by *remembering* its value the first time it's computed, using the remembered value. The term for this technique is *memoization*.

We can encapsulate this idea in a new abstraction called a *thunk*, which is a delayed computation that stores its value when forced. We implement a thunk as a mutable value (a reference) that can be in one of two states: not yet evaluated or previously evaluated. The type definition is

```
 type 'a thunk = 'a thunk_internal ref
         and 'a thunk_internal = | Unevaulated of (unit -> 'a) | Evaluated of 'a ;;  (66)
```

When we need to access the value encapsulated in a thunk, we use the `force` function. It is defined by

```
 let rec force (t :  'a thunk) :  'a =
  match !t with | Evaluated v -> v | Unevaluated f -> t := Evaluated (f ()); force t ;;
                                                                                    (67)
```

17.3.1 The Lazy Module

OCaml provides a module and syntactic sugar for working with lazy computation implemented through thunks, the `Lazy` module.

The type of a delayed computation of an `'a` value is given by `'a Lazy.t`. A delayed computation is specified not by wrapping the expression in `ref (Unevaulated (fun () -> ...))` but by preceding it with the new keyword `lazy`. Forcing a delayed value uses the function `Lazy.force`.

We can consider performing factorials:

```
let fact15 = lazy (print_endline "evaluating 15!"; fact 15) ;;          (68)
```

and forcing the computation, we have

```
Lazy.force fact15 ;;                                                    (69)
```

We can then define infinite streams using the `Lazy` module:

```
type 'a stream_internal = Cons of 'a * 'a stream and 'a stream = 'a stream_internal Lazy.t ;;
                                                                                        (70)
```

17.4 Application: Approximating π

We can evaluate infinite *Taylor series* using lazy computation. Note that

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots \tag{71}$$

We can then write a function to convert a stream of integers to a stream of floats

```
let to_float = smap float_of_int ;;
```
(72)

We can then define a stream of odd integers

```
let odds = smap (fun x -> x * 2 + 1) nats ;;
```
(73)

and a stream of alternating positive and negative ones

```
let alt_signs = smap (fun x -> if x mod 2 = 0 then 1 else -1) nats ;;
```
(74)

and finally, the stream of terms in the π sequence:

```
let pi_stream = smap2 ( /.  )  (to_float (smap (( * ) 4) alt_signs)) (to_float odds) ;;
```
(75)

We can approximate π by takingf the sum of the first few elements, a *partial sum*

```
let pi_approx n = List.fold_left ( +.  )  0.0 (first n pi_stream) ;;
```
(76)

17.5 Problem section: Circuits and boolean streams

17.6 A unit testing framework

17.7 A brief history of laziness

Lazy computation starts with Peter Landlin, observing the relationship between lists and functions.

17.8 Problem set 7: Refs, streams, and music

18 Extension and object-oriented programming

Consider your favorite graphical user interface.

It probably contains *widgets*, buttons, checkboxes, testboxes, radio buttons, etc. The code is organized in a way where each operation is a function and we can organize functions. Instead of adding the same things each time, we can organize the code in a different way where the changes are *localized*. This approach to code organization, organizing by *object* rather than function is referred to as *object-oriented*.

18.1 Drawing graphical elements

We can think of a *scene* as composed of a set of display elements:

```
type display_elt = | Rect of rect | Circle of circle | Square of square ;;     (77)
```

where

- `type rect = {rect_pos : point; rect_width : int; rect_height : int} ;;`
- `type circle = {circle_pos : point; circle_radius : int} ;;`
- `type square = {square_pos : point; square_width : int} ;;`

We will use the Ocaml `Graphics` module. We will rename the module `G` for brevity.

We can draw and transform these shapes, specified on pages 361-364.

18.2 Objects introduced

If we define a data type, an abstraction, `display_elt`, that is a record with a single field called `draw` that stores a drawing function:

```
type display_elt = {draw : unit -> unit} ;;     (78)
```

Then rectangles, circles, squares, and texts are ways of building display elements with the drawing functionality.

Here is an example of a rectangle:

```
let rec (p : point) (w : int) (h : int) : display_elt = { draw = fun () ->
                  G.set_color G.black; G.fill_rect (p.x - w/2) (p.y - h/2) w h } ;;  (79)
```

We can reorganize our code in an *object-oriented* manner by

```
let rect (p : point) (w : int) (h : int) : display_elt =
                    let pos = ref p in let color = ref G.black in
                                 { draw = (fun () ->
      G.set_color (!color); G.fill_rect ((!pos).x - w/2) ((!pos).y - h/2) w h);
             set_pos = (fun p -> pos := p); get_pos = (fun () -> !pos);
            set_color = (fun c -> color := c); get_color = (fun () -> !color) } ;;  (80)
```

18.3 Object-oriented terminology and syntax

The data structure that encapsulates the bits of functionality is an *object*. The various components providing functionality are *methods* and the state variables are *instance variables*.

We create an object by *instantiating* the class, for example, the `circle` class:

```
let circle1 = circle {x = 100; y = 100} 50 ;;                (81)
```

When we make use of a method, we *invoke* the method.

18.4 Inheritance

The code so far violates the edict of irredundancy. To capture commonalities, the object-oriented paradigm allows for definition of a class expressing common aspects, from which both of the classes can *inherit* their behaviors. We refer to the class or class type that is being inherited from as the *superclass* and the inheriting class is the *subclass*.

A class type is given by the constructor

```
class type shape_elt = object ...  end ;;                    (82)
```

The `display_elt` class type can inherit the methods from `shape_elt` by adding an additional `draw` method:

```
class type display_elt = object inherit shape_elt method draw :  unit end ;;      (83)
```

The `inherit` specification works as if the contents of the inherited superclass type were copied into the subclass type at the location in code.

We can add a variable name to the object itself, adding a parenthesized name after the `object` keyword. By convention, we use `this` or `self` and invoke the methods from the superclass with something like `this#get_color`.

18.5 Overriding

Inheritance in OCaml allows subclasses to override the methods in superclasses.

We introduce `method!` where the diacritic marks the method that we are overriding. In this case, we override the superclass's `draw` method.

18.6 Subtyping

We can define a new class type of drawable elements

```
class type drawable = object method draw :  unit end ;;      (84)
```

and we can redefine `draw_list` as

```
let draw_list (d :  drawable list) :  unit = List.iter (fun x -> x#draw) d ;;      (85)
```

where we define `drawable` as a *supertype* of `display_elt`. This is because anything that can be done with drawable can be done with a `display_elt`.

We can use the `:>` operator if we want to view something as its supertype. We can do this because supertypes have a more narrow definition.

18.7 Problem section: Object-oriented counters

18.8 Problem set 8: Force-directed graph drawing

18.9 Problem set 9: Simulating an infectious process

19 Semantics: The environment model

The addition of mutability that enables impure programming paradigms like imperative and procedural programming comes at a cost. The same expression in the same context can evaluate to different values, which makes reasoning more difficult.

19.1 Review of substitution semantics

We recall the abstract syntax of a simple functional language. We also recall a substitution semantics that tells us what to substitute and evaluate. We will develop an environment semantics for the language in two variants: dynamic environment semantics and a lexical environment semantics.

19.2 Environment semantics

In an environment semantics, we directly model a mapping between variables and their values, called an *environment*.

A mapping from elements x, y, z to a, b, c respectively is denoted by $\{x \mapsto a; y \mapsto b; z \mapsto c\}$. This notation evokes OCaml's record notation. We will use E to denote environments (mappings) and primed versions $(E', E'', \dots)$ to denote other environments. Empty environments are notated by $\{\}$ and the environment E augmented to add a mapping $\{x \mapsto v\}$ is denoted by $E\{x \mapsto v\}$.

We will use *Euler's function application notation* $E(x)$ to look up what an environment E maps x to.

19.2.1 Dynamic environment semantics

A substitution semantics is given by a series of rules defining judgements of how expressions evaluate to values. In environment semantics, expressions are not evaluated in isolation. Rather, they are evaluated in the context of an environment that specifies which variable have which values.

We define rules for P evaluating to v in environment E, written as the judgement $E \vdash P \Downarrow v$.

Example 19.1. We can have

$$E \vdash \bar{n} \Downarrow \bar{n}, \qquad E \vdash P + Q \Downarrow E \vdash P \Downarrow \bar{m} | E \vdash Q \Downarrow \bar{n} \Downarrow m \bar{+} n \tag{86}$$

So far, there is no difference from substitution mechanics. The difference is that if we consider `let x = D in B`, it evaluates `B` *in an environment augmented with a new binding of* ***x*** *to its new value* v_D.

In the substitution semantics, we will have substituted away all of the bound variables in a closed expression, so no rule is needed for evaluating variables themselves. But in the environment semantics, since no substitution occurs, we'll need to be able to evaluate expressions that are just variables.

A full list of dynamic environment semantics rules is given on page 397.

There is some subtlety in the dynamic environment semantics. We consider *lexical environment*, the environment in force when the function is defined; and the *dynamic environment*, the environment in force when the function is applied.

The environment semantics presented so far augments the dynamic environment with the new binding induced by application. This manifests a *dynamic environment semantics.*

For consistency with substitution semantics, we should use the lexical environment, manifesting a *lexical environment semantics.*

19.2.2 Lexical environment semantics

We will modify the rules to provide a lexical semantics. The technique is to have functions evaluate to a *package* containing the function and its lexical (defining) environment. This package is called a *closure*. We notate a closure that packages a function P and its environment E as $[E \vdash P]$. In evaluating a function, we construct a closure:

$$E \mapsto \texttt{fun x -> } P \Downarrow [E \mapsto \texttt{fun x -> } P]. \tag{87}$$

We make use of this by

$$E_d \vdash P \;\; Q \Downarrow E_d \vdash \Downarrow [E_l \vdash \texttt{fun x -> } B] | E_d \vdash Q \Downarrow v_Q | E_l\{x \mapsto v_Q\} \vdash B \Downarrow v_B \Downarrow v_B \tag{88}$$

Rather than augment the dynamic environment E_d in evaluating the body, we augment the lexical environment E_l extracted from the closure.

19.3 Conditionals and booleans

Recall in section 13.5, exercises asked us to develop abstract syntax and substitution semantics rules for booleans and conditionals. We will want similar rules for environment semantics.

19.4 Recursion

The dynamic environment semantics allows for recursion due to its dynamic nature.

The lexical semantics does not benefit from ill-formed recursive functions.

19.5 Implementing environment semantics

In 13.4.2, we presented an implementation of the substitution semantics in the form of a function `eval : expr -> expr`. Modifying it to follow the environment semantics requires a few simple changes.

Note that the evaluation is relative to an environment so the `eval` function should take an additional argument of type `env` for environment. Under lexical environment semantics require expressions to evaluate to values that include more than the pertinent subset of expressions — expressions evaluate to closures so we need an extended notion of value in a type `value`.

We can define

```
type env = (varid * value ref) list ;;
```
(89)

and the value type

```
type value = | Val of expr | Closure of (expr * env) ;;
```
(90)

19.6 Semantics of mutable storage

We will expand our lexical environment semantics to allow for imperative programming with references and assignment.

We add a *location*, which is an index or pointer to an abstract model of memory that we call the *store*. A store S is a finite mapping from locations to values.

Evaluations thus are relative to a store in addition to an environment so judgements look like $E, S \vdash P \Downarrow \cdots$.

A semantic rule for references is

$$E, S \vdash \texttt{ref } P \Downarrow E, S \vdash P \Downarrow v_P, S' \Downarrow l, S'\{l \mapsto v_P\}. \tag{91}$$

According to this rule, to evaluate an expression of the form `ref` P in environment E and store S, we evaluate P in the environment and store, yielding value v_P for P and a new store S' (for side effects to S in evaluation). The value for the reference is a new location l and a new store S' augmented so that l maps to v_P.

We also define another rule for assignment to a reference, i.e. expressions like P`:=` Q, which involves evaluating P to a location l, evaluating Q to a value v_Q, and updating the store so l maps to v_Q. The rule is

$$E, S \vdash P\texttt{:=}Q \Downarrow E, S \vdash P \Downarrow l, S' | E, S' \vdash Q \Downarrow v_Q, S'' \Downarrow (), S''\{l \mapsto v_Q\}. \tag{92}$$

The full set of lexical environment semantics rules are on pages 409-410.

19.6.1 Lexical environment semantics of recursion

The extended language with references and assignment is sufficient to provide a semantics for the `let rec` construct. The rule is

$$\begin{aligned} E, S \vdash \texttt{let rec}\ x = D\ \texttt{in}\ B \Downarrow E\{x \mapsto l\}, S\{l \mapsto \texttt{unassigned}\} \vdash D[x \mapsto \texttt{!}x] \Downarrow v_D, S' \\ | E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash B[x \mapsto \texttt{!}x] \Downarrow v_B, S'' \Downarrow v_B, S'' \end{aligned} \tag{93}$$

as desired.

20 Final project: Implementing MiniML

The final project is the implementation of a small subset of an OCaml-like language.

20.1 Overview

The language we will implement includes a small subset of constructs and has limited support for types.

We will implement a *Turing-complete* OCaml subset known as MiniML, in the form of an interpreter for expressions of the language written in OCaml. This is a *metacircular interpreter.*

The task is divided into three sections: substitution model, dynamic scoped environment model, and extensions.

20.1.1 Grading and collaboration

Projects are done individually under standard rules of collaboration. The final project will be graded on correctness of implementation of the first two stages; design and style of submitted code; and scope of project (including extensions).

20.2 Implementing a substitution semantics for MiniML

The abstract syntax is given by a type definition on page 418. It is also available in `expr.ml`.

The details of the other files can be found on page 419.

We are asked to complete stages 1-5.