编程语言的设计原理

Design Principles of Programming Languages

Zhenjiang Hu, Yingfei Xiong, Haiyan Zhao,

胡振江　熊英飞　赵海燕

Peking University, Spring, 2018

Chapter 0+:   Implementation

A quick tour of OCaml

Utilities in Ocaml system

An Implementation for Arithmetic Expression

A Quick Tour of OCaml

Resources

- Overview
  - http://ocaml.org/learn/tutorials/basics.html
- Tutorials
  - http://ocaml.org/learn/tutorials/
- Download
  - http://caml.inria.fr/download.en.html
- MOOC
  - https: //www.fun-mooc.fr/courses/parisdiderot/56002/session01/about

Why OCaml?

What we learn in this course, is mostly *conceptual* and *mathematical*. However:

- Some of the ideas are *easier to grasp* if you can *see them work*;
- Experimenting with small implementations of programming languages is an excellent way to *deepen intuitions.*

OCaml language is chosen for these purposes

- General programming language with an emphasis on *expressiveness* and *safety*.

OCaml used in the Course

Concentrates just on the "***core***" of the language, *ignoring* most of its features, like modules or objects. For

- some of the ideas in the course are *easier to grasp* if you can "*see them work*"
- *experimenting with small implementations* of programming languages is an excellent way to deepen intuitions

Quick fact sheet

- Some facts about Caml (*Categorical Abstract Machine Language*)
  - Created in 1987 by INRIA - France's national research institute for computer science (Haskell 1.0 is from 1990)
  - Originated from ML but was intended for in house projects of INRIA
  - Short timeline:

    Caml (1987) → Caml Light (1990) → OCaml (1995)
  - Currently at version 4.06.0 (released in Nov, 2017)

OCaml

A large and powerful language (*safety* and *reliability*)

- the most popular variant of the Caml language
  - Collaborative Application Markup Language ? (协作应用程序标记语言)
- extending the core Caml language with
  - a fully-fledged object-oriented layer
  - powerful module system
  - a sound, polymorphic type system featuring type inference.
- *a functional programming language*
  - i.e., a language in which the functional programming style is the dominant idiom

OCaml system is *open source software*

Functional Programming

Functional style can be described as a combination of...

- *persistent* data structures (which, once built, are never changed)
- *recursion* as a primary control structure
- heavy use of *higher-order* functions (that take functions as arguments and/or return functions as results)

Imperative languages, by contrast, emphasize...

- *mutable* data structures
- *looping* rather than *recursion*
- *first-order* rather than *higher-order* programming (though many object-oriented design patterns involve higher-order idioms — e.g., Subscribe/Notify, Visitor, etc.

The Top Level

Ocaml, as most functional programming implementation, provides both an *interactive top level* and a *compiler* that produces standard executable binaries.

- The top level provides a *convenient way* of experimenting with small programs.

The mode of interacting with the top level is *typing in a series of expressions*; OCaml *evaluates them* as they are typed and *displays the results* (and their types). In the interaction ,

- lines beginning with **#** are inputs
- lines beginning with **-** are the system's responses.

Note that inputs are always terminated by a

double semicolon ;;

Expressions

OCaml is an *expression language*.

A program is *an expression*.

The "*meaning*" of program is the *value* of the expression.

```
# 16 + 18;;
- : int = 34

# 2*8 + 3*6;;
- : int = 34
```

Every expression has *exactly one type* (*no pure command*, even assignment, unit)

When an expression is evaluated, *one of 4 things* may happen:

1. It may evaluate to ***a value*** of the same type as the expression.
2. It may raise ***an exception*** (discussed later)
3. It may ***not terminate***
4. It may ***exit***.

Expressions

```
#  if  1 < 2 then  1 else 1.6;;
```

What will happen?

```
#  if  1 < 2 then  1 else 1.6;;
                          ^^^
Error:  This expression has type float but an expression was
expected of type int
```

In general, the compiler doesn’t try to figure out the value of the *test* during type checking. Instead, it requires that *both branches* of the conditional have the same type (no matter how the test turns out).

Basic types

Include: unit, int, char, float, bool, and string

- char: ‘a’, ‘\120’ (decimal, ‘x’)
- string : a built-in type, unlike C, “hello” , “”, s.[i]
- bool : logical operators && , || are *short-circuit* version

Strongly typed language (not like the weakly-typed C)

- Every expression must have a type, and expressions of one type may not be used as expressions in another type
- There are *no implicit coercions (casting)* between types in Ocaml !!
  - int_of_float, float_of_Int, …….

Basic types

Ocaml type	**Range**
int	31-bit signed int (roughly +/- 1 billion) on 32-bit processors, or 63-bit signed int on 64-bit processors
float	IEEE double-precision floating point, equivalent to C's double
bool	A boolean, written either *true* or *false*
char	An 8-bit character . Not support Unicode or UTF-8, a serious flaw in OCaml.
string	A string . Strings are not just lists of characters. They have their own, more efficient internal representation.
unit	Written as ()

Type boolean

There are *only two values* of type bool: *true* and *false*.

Comparison operations return boolean values.

```
# 1 = 2;;
- : bool = false

# 4 >= 3;;
- : bool = true
```

not is a unary operation on booleans

```
# not (5 <= 10);;
- : bool = false
# not (2 = 2);;
- : bool = false
```

Conditional expressions

The result of the conditional expression if B then E1 else E2 is either the result of E1 or that of E2, depending on whether the result of B is *true* or *false*.

```
# if 3 < 4 then 7 else 100;;
- : int = 7
# if 3 < 4 then (3 + 3) else (10 * 10);;
- : int = 6
# if false then (3 + 3) else (10 * 10);;
- : int = 100
# if false then false else true;;
- : bool = true
```

Giving things names

The **let construct** gives a *name* to the result (*value*) of an expression so that it can be used later.

let name = expr

```
# let inchesPerMile = 12*3*1760;;
val inchesPerMile : int = 63360

# let x = 1000000 / inchesPerMile;;
val x : int = 15
```

Variables are *names* for *values.*

Names may contain *letters (*upper & lower case*), digits, _, and the ’,* and must begin with a lowercase letter or underscore.

Giving things names

Definition using **let** can be nested using the **in** form.

let name = expr1 **in** expr2

expr2 is called the *body* of **let,** name is defined as the value of expr1 within the body

```
# let x = 1 in
    let x = 2 in
    let y = x + x  in
       x + y ;;
- : int = 6
```

The scope of x?

Giving things names

```
# let x = 1;;
val x : int =1
# let  z  =
    let x = 2 in
    let x = x + x  in
       x + x ;;
val z : int = 8
# x;;
- : int = 1
```

Binding is *static*: if there is *more than* one definition for a variable, the value of the variable is defined by the most recent **let** definition for it.

The variable is bound only in the *body* of **let**.

Functions

```
# let cube (x: int) = x*x*x;;
val cube : int -> int = <fun>
# cube 9;;
- : int = 729
```

We call x the *parameter* of the function *cube*; the expression x*x*x is its *body*. The expression cube 9 is an *application* of *cube* to the argument *9*. (How about C/C++?)

Here, **int->int** (pronounced “int arrow int”) indicates that *cube* is a function that should be applied to an ***integer argument*** and that ***returns an integer***.

Note that OCaml responds to a function declaration by printing just <fun> as the function’s *value*.

The precedence of function application is *higher* than most operators.

Functions

A function with *two parameters*:

```
# let sumsq (x: int) (y: int) = x*x + y*y;;
val sumsq : int -> int -> int = <fun>

# sumsq 3 4;;
- : int = 25
```

The type printed for sumsq is int->int->int, indicating that it should be applied to ***two integer arguments*** and yields ***an integer*** as its ***result***.

Note that the syntax for invoking function declarations in OCaml is *slightly different from* languages in the C/C++/Java family:

use cube 3 and sumsq 3 4 rather than cube(3) and sumsq(3, 4), since *multiple-parameter* functions are implemented as *nested* functions (called *Currying*)

Recursive functions

We can translate *inductive definitions* directly into *recursive functions*.

```
# let rec sum(n:int) = if n = 0 then 0 else n + sum(n-1);;
val sum : int -> int = <fun>
# sum 6;;
- : int = 21

# let rec fact(n:int) = if n = 0 then 1 else n * fact(n-1);;
val fact : int -> int = <fun>
# fact 6;;
- : int = 720
```

The rec after the let tells Ocaml that this is a *recursive function* — one that needs to *refer to itself* in its own body.

What will happen if dropping the rec?

Recursive functions

```
# let rec power k x = if k = 0 then 1.0 else  x *. (power (k-1) x) ;;
val power : int -> float -> float = <fun>
# power 5 2.0; ;
 -: float = 32
```

```
# let b_power k x = (float_of_int k) *. x;;
val b_power : int -> float -> float = <fun>
# let b_power k x = if k = 0 then 1.0 else  x *. (b_power (k-1) x) ;;
val b_power : int -> float -> float = <fun>
# b_power 5 2.0; ;
 -: float = ?
 -: float = 16
```

Recursive functions: Making change

Another example of recursion on integer arguments:

Suppose a bank has an “infinite” supply of coins (*pennies*, *nickles*, *dimes*, and *quarters*, and *silver dollars*), and it has to give a customer a certain sum. *How many ways* are there of doing this?

For example, there are *4 ways of making change* for 12 cents:

- 12 pennies
- 1 nickle and 7 pennies
- 2 nickles and 2 pennies
- 1 dime and 2 pennies

We want to write a function change that, when applied to *12*, returns *4*.

Recursive functions: Making change

Let’s first consider a simplified variant of the problem where the bank only has one kind of coin: *pennies*.

In this case, there is *only one way* to make change for a given amount: pay the whole sum in pennies!

```
# (* No. of ways of paying a in pennies *)
let rec changeP (a: int) = 1;;
```

That wasn’t very hard.

Note: *Comments* starts with **(*** and end with ***)**

Recursive functions: Making change

Now suppose the bank has both *nickels* and *pennies*.

If *a* is less than *5* then we can only pay with *pennies*. If not, we can do *one of two things*:

- Pay in *pennies*; we already know how to do this.
- Pay with at least one *nickel*. The number of ways of doing this is the number of ways of making change (with *nickels* and *pennies*) for a-5.

```
# (* No. of ways of paying in pennies and nickels *)
let rec changePN (a:int) =
    if a < 5 then changeP a
    else changeP a + changePN (a-5);
```

Recursive functions: Making change

Continuing the idea for *dimes* and *quarters*:

```
# (* ... pennies, nickels, dimes *)
let rec changePND (a:int) =
    if a < 10 then changePN a
    else changePN a + changePND (a-10);;

# (* ... pennies, nickels, dimes, quarters *)
let rec changePNDQ (a:int) =
    if a < 25 then changePND a
    else changePND a + changePNDQ (a-25);;
```

Recursive functions: Making change

```
# (* Pennies, nickels, dimes, quarters, dollars *)
    let rec change (a:int) =
        if a < 100 then changePNDQ a
        else changePNDQ a + change (a-100);;
```

Recursive functions: Making change

Some tests:

```
# change 5;;
- : int = 2
# change 9;;
- : int = 2
# change 10;;
- : int = 4
# change 29;;
- : int = 13
# change 30;;
- : int = 18
# change 100;;
- : int = 243
# change 499;;
- : int = 33995
```

Aggregate types

OCaml provides a rich set of aggregate types for storing a collection of data values, including

- lists
- tuples
- disjoint union (also called tagged unions, or variant records)
- records,
- arrays

Lists

One handy structure for storing a collection of data values is a list.

- provided as a *built-in* type in OCaml and a number of other popular languages (e.g., Lisp, Scheme, and Prolog—but not, unfortunately, Java), used *extensively* in FP programs.
- a sequence of values of the same type.
- built in OCaml by writing out its elements, enclosed in *square brackets* and separated by *semicolons*.

```
# [1; 3; 2; 5];;
- : int list = [1; 3; 2; 5]
```

The type printed for this list is pronounced either "integer list" or "list of integers".

The *empty list*, written [], is sometimes called "nil."

Lists are homogeneous

OCaml does not allow different types of elements to be mixed within the same list:

```
# [1; 2; "dog"];;
Characters 7-13:
Error:   This expression has type string but an
expression was expected of type int
```

Constructing Lists

OCaml provides a number of built-in operations that return lists.

The *most basic one* creates a new list by adding an element to the front of an existing list.

- written :: and pronounced "*cons*" (for it constructs lists).

```
# 1 :: [2; 3];;
- : int list = [1; 2; 3]

# let add123 (l: int list) = 1 :: 2 :: 3 :: l;;
val add123 : int list -> int list = <fun>

# add123 [5; 6; 7];;
- : int list = [1; 2; 3; 5; 6; 7]

# add123 [];;
- : int list = [1; 2; 3]
```

Constructing Lists

Any list can be built by “*consing*” its elements together:

```
# 1 :: 2 :: 3 :: 2 :: 1 :: [] ;;;
 : int list = [1; 2; 3; 2; 1]
```

In fact, [e_1; e_2; . . . ; e_n] is simply a *shorthand* for

e_1 :: e_2 :: . . . :: e_n :: []

Note that, when omitting parentheses from an expression involving several uses of ::, we *associate to the right*

- i.e., 1::2::3::[] means the same thing as 1::(2::(3::[]))
- By contrast, arithmetic operators like + and - associate to the left: 1-2-3-4 means ((1-2)-3)-4.

Taking Lists Apart

OCaml provides *two basic operations* for extracting the parts of a list (i.e., *deconstruction*).

- List.hd (*pronounced* “head”) returns the *first element* of a list.

  ```
  # List.hd  [1; 2; 3];;
  - : int = 1
  ```

- List.tl (*pronounced* “tail”) returns *everything but the first element*.

  ```
  # List.tl [1; 2; 3];;
  - : int list = [2; 3]
  ```

More list examples

```
# List.tl (List.tl [1; 2; 3]);;
- : int list = [3]

# List.tl (List.tl (List.tl [1; 2; 3]));;
- : int list = []

# List.hd (List.tl (List.tl [1; 2; 3]));;
- : int = 3
```

Recursion on lists

Lots of useful functions on lists can be written using *recursion*.

- Here’s one that sums the elements of a list of numbers:

```
# let rec listSum (l: int list) =
    if l = [] then 0
    else List.hd l + listSum (List.tl l);;
 val listSum : int list -> int = <fun>

# listSum [5; 4; 3; 2; 1];;
- : int = 15
```

Consing on the right

```
# let rec snoc (l: int list) (x: int) =
    if l = []  then x::[]
    else List.hd l :: snoc(List.tl l) x;;
val snoc : int list -> int -> int list = <fun>

# snoc [5; 4; 3; 2] 1;;
- : int list = [5; 4; 3; 2; 1]
```

A better rev

```
# (* Adds the elements of l to res in reverse order *)
let rec revaux (l: int list) (res: int list) =
    if l = [] then res
    else revaux (List.tl l) (List.hd l :: res);;
val revaux : int list -> int list -> int list = <fun>

# revaux [1; 2; 3] [4; 5; 6];;
- : int list = [3; 2; 1; 4; 5; 6]

# let rev (l: int list) = revaux l [];;
val rev : int list -> int list = <fun>
```

Tail recursion

It is usually fairly easy to rewrite a recursive function in *tail-recursive style*.

- e.g., the usual factorial function is not *tail recursive* (because one multiplication remains to be done after the recursive call returns):

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

It can be transformed into a tail-recursive version by performing the multiplication before the recursive call and passing along a separate argument in which these multiplications “accumulate”:

```
# let rec factaux (acc:int) (n:int) =
      if n = 0 then acc
      else factaux (acc*n) (n-1);;

# let fact (n:int) = factaux 1 n;;
```

Basic Pattern Matching

Recursive functions on lists tend to have *a standard shape*:

- test whether the list is *empty*, and if it is not
- do something involving the head element and the tail.

```
# let rec listSum (l:int list) =
        if  l = [] then 0
        else List.hd l + listSum (List.tl l);;
```

OCaml provides a convenient *pattern-matching* construct that bundles the emptiness test and the extraction of the head and tail into *a single syntactic form*:

```
# let rec listSum (l: int list) =
match l with
        [] -> 0
      | x::y -> x + listSum y;;
```

Basic Pattern Matching

Pattern matching can be used with *types* other than lists, like other *aggregate types*, and even *simple types*.

For example, here it is used on integers:

```
# let rec fact (n:int) =
    match n with
          0  -> 1
       | _  -> n * fact(n-1);;
```

here _ pattern is a *wildcard* that matches any value.

Complex Patterns

The basic elements (*constants*, *variable binders*, *wildcards*, [], ::, etc.) may be combined in arbitrarily complex ways in match expressions:

```
# let silly  l =
    match l with
        [_; _; _]             ->  "three elements long"
      | _::x::y::_::_::rest  ->  if x > y then "foo" else "bar"
      | _                     -> "dunno";;
val silly : int list -> string = <fun>
# silly [1; 2; 3];;
- : string = "three elements long"
# silly [1; 2; 3; 4];;
- : string = "dunno"
# silly [1; 2; 3; 4; 5];;
- : string = "bar"
```

Type Inference

One pleasant feature of OCaml is its powerful *type inference mechanism* that allows the compiler to *calculate the types of variables* from the way in which they are used.

```
# let rec fact n =
      match n with
           0          ->  1
       |   _          ->  n * fact (n - 1);;
val fact : int -> int = <fun>
```

The compiler can tell that fact takes an *integer* argument because *n* is used as an argument to the integer * and - functions.

Type Inference

Similarly:

```
# let rec listSum l   =
      match l with
            []              ->  0
         |   x::y           ->  x + listSum y;;
val listSum : int list -> int = <fun>
```

Polymorphism (first taste)

```
# let rec length l  =
     match l with
           []       ->  0
      |      _::y  ->  1 + length y;;
val length  :  'a list -> int = <fun>
```

- The 'a in the type of length, pronounced "alpha," is a *type variable* standing for an *arbitrary type*.
- The inferred type tells us that the function can take a list with elements of *any type* (i.e., a list with elements of type *alpha*, for any choice of alpha).

Tuples

Items connected by *commas* are “tuples.” (The enclosing parenthesis are optional)

```
# "age", 38;;
- : string * int = "age", 38

# "professor", "age", 33;;
- : string * string * int = "professor", "age", 33

# ("children", ["bob";"ted";"alice"]);;
- : string * string list =  "children", ["bob"; "ted"; "alice"]

# let g (x, y) = x*y;;
val g : int * int -> int = <fun>
```

Tuples are not lists

Do not confuse them!

```
# let tuple = "cow", "dog", "sheep";;
val tuple : string * string * string =  "cow", "dog", "sheep"

# List.hd tuple;;
Error: This expression has type string * string * string
       but an expression was expected of type 'a list
```

```
# let tup2 = 1, "cow";;
val tup2 : int * string = 1, "cow"

# let l2 = [1; "cow"];;
Error: This expression has type string but an expression was
expected of type  int
```

Tuples and pattern matching

Tuples can be “deconstructed” by pattern matching, like list:

```
# let lastName name =
      match name with
          (n, _, _)  ->  n;;

# lastName (“Zhao", “Haiyan", “PKU");;
- : string = “Zhao"
```

Example: Finding words **

Suppose we want to take a *list of characters* and return a *list of lists of characters*, where each element of the final list is a “word” from the original list.

```
# split [’t’;’h’;’e’;’ ’;’b’;’r’;’o’;’w’;’n’; ’ ’;’d’;’o’;’g’];;
- : char list list = [[’t’; ’h’; ’e’]; [’b’; ’r’; ’o’; ’w’; ’n’];
            [’d’; ’o’; ’g’]]
```

(Character constants are written with single quotes.)

An implementation of split

```
# let rec loop w l =
      match l with
           []          ->  [w]
        |  (’ ’::ls)  ->  w :: (loop [] ls)
        |  (c::ls)    ->  loop (w @ [c]) ls;;
val loop : char list  -> char list  -> char list list  =  <fun>

# let split l = loop [] l;;
val split : char list -> char list list = <fun>
```

Note the use of both *tuple patterns* and *nested patterns*.

The @ operator is shorthand for List.append.

Aside: Local function definitions

The loop function is *completely local* to **split**: there is no reason for anybody else to use it — or even for anybody else to be able to see it!

It is good style in OCaml to write such definitions *as local bindings*:

```
# let split l =
      let rec loop w l =
            match l with
               []          ->      [w]
            | (’ ’::ls)  ->      w :: (loop [] ls)
            | (c::ls)    ->      loop (w@[c]) ls
      in loop [] l;;
```

Local function definitions

In general, any *let definition* that can appear at the top level

```
# let  ...  ;;
# e;;
```

can also appear in a let ... in ... form

```
# let ... in e;;
```

A Better Split ?

Our split function worked fine for the examples we tried it on so far.   But here are some other tests:

```
# split  [’a’; ’ ’; ’ ’; ’b’];;
- : char list list = [[’a’]; []; [’b’]]

# split [’a’; ’ ’];;
- : char list list = [[’a’]; []]
```

Could we refine split so that it would leave out these spurious empty lists in the result?

A Better Split

Sure. First rewrite the *pattern match* a little (without changing its behavior)

```
# let split l =
      let rec loop w l =
            match w, l with
                 _, []            ->        [w]
               | _, (’ ’::ls)     ->        w :: (loop [] ls)
               | _, (c::ls)       ->        loop (w@[c]) ls
      in loop [] l;;
```

A Better Split

Then add a couple of clauses:

```
# let better_split l =
      let rec loop w l =
            match w, l with
               [], []            ->        []
             | _,  []            ->        [w]
             | [],  (’ ’::ls)    ->        loop [] ls
             | _,  (’ ’::ls)     ->        w :: (loop [] ls)
             | _,  (c::ls)       ->        loop (w@[c]) ls
      in loop [] l;;

# better_split [’a’; ’b’; ’ ’; ’ ’; ’c’; ’ ’; ’d’; ’ ’];;
- : char list list = [[’a’; ’b’]; [’c’]; [’d’]]
# better_split [’a’; ’ ’];;
- : char list list = [[’a’]]
# better_split [’ ’; ’ ’];;
- : char list list = []
```

Basic Exceptions

OCaml's *exception mechanism* is roughly similar to that found in, for example, Java. It begins by defining an exception:

```
# exception Bad;;
```

Now, encountering `raise Bad` will immediately *terminate evaluation* and return control to the top level:

```
# let rec fact n =
      if n<0 then raise Bad
      else if n=0 then 1
      else n * fact(n-1);;
# fact (-3);;
Exception: Bad.
```

(Not) catching exceptions

Naturally, exceptions can also be caught within a program (using the try … with … form), by pattern matching

```
try e with
      p_1    ->       e_1
     |p_2    ->       e_2
      ……
     | p_n   ->       e_n
```

Exceptions are used in Ocaml as a *control mechanism*, either to signal errors, or *to control the flow of execution*.

- When an exception is raised, the current execution is aborted, and control is thrown to the most recently entered active exception handler.

Defining New Types of Data

Predefined types

We have seen a number of data types:

int
$bool$
$string$
$char$
$[x; y; z]\ lists$
$(x, y, z)\ tuples$

Ocaml has a number of *other built-in data types* — in particular, float, with operations like +., *., etc.

One can also create completely *new data types*.

The need for new types

The ability to construct new types is an *essential part* of most programming languages.

For example, suppose we are building a (very simple) graphics program that displays *circles* and *squares*. We can represent each of these with *three real numbers* ...

The need for new types

A *circle* is represented by the coordinates of its *center* and its *radius*. A *square* is represented by the coordinates of its bottom *left corner* and its *width*.

- both *shapes* can be represented as elements of the type:

```
float * float * float
```

Two problems with using this type to represent *circles* and *squares*.

- A bit long and unwieldy, both to write and to read.
- Prone to mix circles and squares since their types are identical, might accidentally apply the areaOfSquare function to a circle and get a nonsensical result.

```
# let areaOfSquare (_,  _, d)  =  d *. d;;
```

Data Types

We can improve matters by defining **square** as *a new type*:

```
# type square = Square of float * float * float;;
```

This does two things:

- creates a *new type* called square that is different from any other type in the system.
- creates a *constructor* called Square (with a capital S) that can be used to create a square from three floats.

```
# Square (1.1, 2.2, 3.3);;
- : square = Square  (1.1, 2.2, 3.3)
```

Taking data types apart

And taking types apart with (surprise, surprise, ...) *pattern matching*

```
# let areaOfSquare s =
      match s with
          Square(_, _, d)  ->  d *. d;;
val areaOfSquare : square -> float  =  <fun>

# let bottomLeftCoords  s =
      match s with
          Square(x, y, _)  -> (x, y);;
val bottomLeftCoords : square -> float * float  =  <fun>
```

constructors like Square can be used both as *functions* and as *patterns.*

Taking data types apart

These functions can be written a little more concisely by combining the *pattern matching* with the *function header*:

```
# let areaOfSquare (Square (_, _, d)) = d *. d;;
# let bottomLeftCoords (Square (x, y, _)) = (x, y);;
```

Variant types

Back to the idea of a graphics program, we want to have several *shapes* on the screen at once. To do this we probably want to keep a list of *circles* and *squares*, but such a list would be heterogenous. How do we make such a list?
Answer: Define a type that can be *either* a *circle* *or* a *square*.

```
# type shape =     Circle of float * float * float
                  | Square of float * float * float;;
```

Now *both constructors* Circle and Square create values of type shape.

```
# Square (1.0, 2.0, 3.0);;
- : shape = Square (1.0, 2.0, 3.0)
```

A type that can have *more than one form* is often called a *variant type*.

Pattern matching on variants

We can also write functions that ***do the right thing*** on all forms of a *variant type*, by using *pattern matching*:

```
# let area s =
      match s with
            Circle (_, _, r)     ->      3.14159 *. r *. r
         |  Square (_, _, d)  ->      d *. d;;

# area (Circle (0.0, 0.0, 1.5));;
- : float = 7.0685775
```

Variant types

A heterogeneous list:

```
# let l  =  [Circle (0.0, 0.0, 1.5);
             Square (1.0, 2.0, 1.0);
             Circle (2.0, 0.0, 1.5);
             Circle (5.0, 0.0, 2.5)];;

# area (List.hd l);;
- : float = 7.0685775
```

Data Type for Optional Values

Suppose we are implementing a simple lookup function for a telephone directory. We want to give it a *string* and get back a *number* (say an integer), i.e, a function whose type is:

```
        lookup: string -> directory -> int
```

where directory is a (yet to be decided) type that we'll use to represent the directory.

However, this isn't quite enough. What happens if a given string isn't in the directory? What should lookup return?

There are several ways to deal with this issue. One is to raise an *exception*. Another uses the following data type:

```
# type optional_int = Absent | Present of int;;
```

Data Type for Optional Values

To see how this type is used, let's represent our directory as a list of *pairs*:

```
# let directory = [ ("Joe", 1234);  ("Martha", 5672);
                    ("Jane", 3456);  ("Ed", 7623)];;
# let rec lookup s l =
      match l with
          []          ->        Absent
        |  (k, i)::t  ->        if k = s then Present(i)
                                else lookup s t;;

# lookup "Jane" directory;;
-   : optional_int = Present 3456

# lookup "Karen" directory;;
- : optional_int = Absent
```

Built-in options

options are often useful in functional programming, OCaml provides a *built-in type* t option for each type t. Its constructors are None (corresponding to Absent) and Some (for Present).

```
# let rec lookup s l =
      match l with
              []          ->      None
          |   (k,i)::t  ->        if k = s then Some(i)
                                  else lookup s t;;

# lookup "Jane" directory;;
- : optional_int = Some 3456
```

Enumerations

The option type has one variant, None, that is a "*constant*" constructor *carrying no data values with it*. Data types in which *all* the variants are constants can actually be quite useful ...

```
# type color = Red | Yellow | Green;;
# let next c =
      match c with Green -> Yellow | Yellow -> Red | Red -> Green;

# type day =  Sunday | Monday | Tuesday | Wednesday
                  | Thursday | Friday | Saturday;;
# let weekend d =
     match d with
          Saturday ->  true
        |  Sunday   ->  true
        |  _            ->  false;;
```

A Boolean Data Type

A simple data type can be used to replace the built-in booleans, by using the constant constructors True and False to represent *true* and *false* . Here use different names as needed to avoid confusion between our booleans and the built-in ones:

```
# type myBool = False | True;;
# let myNot b =  match b with False -> True | True -> False;;
# let myAnd b1 b2  =
        match (b1, b2) with
                (True, True)            -> True
            |   (True, False)           -> False
            |   (False, True)           -> False
            |   (False, False)          -> False;;
```

Note that the behavior of myAnd is not quite the same as the built-in &&!

Recursive Types

Consider the *tiny language of arithmetic expressions* defined by the following (BNF-like) grammar:

```
exp   ::=     number
            | ( exp + exp )
            | ( exp - exp )
            | ( exp * exp )
```

Recursive Types

This grammar can be translated directly into a data type definition:

```
# type  ast  =
        ANum of int
      |  APlus of  ast * ast
      |  AMinus of ast * ast
      |  ATimes of ast * ast ;;
```

Notes:

- This datatype (like the original grammar) is *recursive*.
- The type ast represents *abstract syntax trees*, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses

An evaluator for expressions

write an evaluator for these expressions：

```
val eval : ast -> int = <fun>


# eval (ATimes (APlus (ANum 12, ANum 340), ANum 5));;
- : int = 1760
```

An evaluator for expressions

The solution uses a *recursive function* plus a *pattern match*.

```
# let rec eval e =
    match e with
         ANum i                  ->  i
      |  APlus (e1, e2)          ->  eval e1 + eval e2
      |  AMinus (e1, e2)         ->  eval e1 - eval e2
      |  ATimes (e1, e2)         ->  eval e1 * eval e2;;
```

Polymorphism

Polymorphism

We encountered the concept of *polymorphism* very briefly. Let’s review it in a bit more detail

```
# let rec last l =
      match l with
           []          -> raise Bad
        |  [x]         -> x
        |  _::y        -> last y
```

What type should we give to the parameter l?

It doesn’t matter what type of objects are stored in the list: int list or bool list. However, if we chose one of these types,   would not be able to apply *last* to the other.

Polymorphism

Instead, we can give l the type ’a list, standing for an arbitrary type.

Ocaml will figure out what type we need when we use it.

This version of last is said to be polymorphic, because it can be applied to many *different types* of arguments. (“Poly” = many, “morph” = shape.)

In other words,

last : ’a list -> ’a

can be read as *“last is a function that takes a list of elements of any type ‘a and returns an element of ‘a.”*

Here, the type of the elements of l is ’a. This is a ***type variable***, which can *instantiated* each time we apply last, by replacing ’a with any type that we like.

A polymorphic append

```
# let rec append (l1: ’a list) (l2: ’a list) =
      if l1 = [] then l2
      else List.hd l1 :: append (List.tl l1) l2;;
val append : ’a list -> ’a list -> ’a list = <fun>

# append [4; 3; 2] [6; 6; 7];;
- : int list = [4; 3; 2; 6; 6; 7]

# append ["cat"; "in"] ["the"; "hat"];;
- : string list = ["cat"; "in"; "the"; "hat"]
```

Programming With Functions

Functions as Data

Functions in OCaml are *first class citizen* — they have the *same rights and privileges* as *values* of any other types, e.g., they can be

- passed as arguments to other functions,
- returned as results from other functions,
- stored in data structures such as tuples and lists,
- etc.

map: “apply-to-each”

OCaml has a predefined function List.map that takes a function f and a list l and ***produces another list*** by applying f to each element of l.

First let’s look at some examples.

```
# List.map square [1; 3; 5; 9; 2; 21];;
- : int list = [1; 9; 25; 81; 4; 441]

# List.map not [false; false; true];;
- : bool list = [true; true; false]
```

Note that List.map is polymorphic: it works for lists of *integers*, *strings*, *booleans*, etc.

More on map

An interesting feature of List.map is *its first argument* is itself *a function*. For this reason, we call List.map a *higher-order* function.

Natural uses for *higher-order functions* arise frequently in programming. One of OCaml's **strengths** is that it makes *higher-order functions very easy to work with*.

In other languages such as Java, higher-order functions can be (and often are) simulated using objects.

filter

Another useful higher-order function is List.filter: when applied to a list l and a boolean function p, it builds a list of the elements from l for which p returns true.

```
# let rec even (n:int)  =
        if n=0 then true else if n=1 then false
        else if n<0 then even (-n)  else even (n-2);;
val even : int -> bool = <fun>

# List.filter even [1; 2; 3; 4; 5; 6; 7; 8; 9];;
- : int list = [2; 4; 6; 8]

# List.filter palindrome  [[1]; [1; 2; 3]; [1; 2; 1]; []];;
- : int list list = [[1]; [1; 2; 1]; []]
```

Defining map

List.map comes predefined in the OCaml system, but there is nothing magic about it : we can easily define our own map function with the same behavior.

```
# let rec map (f: ’a -> ’b) (l: ’a list) =
      if l = [] then []
      else f (List.hd l) :: map f (List.tl l)
val map : (’a -> ’b) -> ’a list -> ’b list = <fun>
```

The type of map is probably even more polymorphic than you expected! The list that it returns can actually be of a different type from its argument:

```
# map String.length ["The"; "quick"; "brown"; "fox"];;
- : int list = [3; 5; 5; 3]
```

Defining filter

Similarly, we can define our own filter that behaves the same as List.filter.

```
# let rec filter (p: ’a -> bool) (l: ’a list) =
      if l = [] then  []
      else if p (List.hd l) then
            List.hd l :: filter p (List.tl l)
      else
            filter p (List.tl l)
val filter : (’a -> bool) -> ’a list -> ’a list  = <fun>
```

Multi-parameter functions

We have seen two ways of writing functions with multiple parameters:

```
# let foo x y = x + y;;
val foo : int -> int -> int = <fun>

# let bar (x, y) = x + y;;
val bar : int * int -> int = <fun>
```

Another useful higher-order function: fold

```
# let rec fold f l acc =
      match l with
          []        ->        acc
        | a::l      ->        f a (fold f l acc);;
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
# fold (fun a b -> a + b) [1; 3; 5; 100]  0;;
- : int = 109
```

In general:

$$f\ [a_1; \ldots; a_n]\ b$$

is

$$f\ a_1\ (f\ a_2\ (\ldots\ (f\ a_n\ b)\ \ldots)).$$

Using fold

Most of the list-processing functions we have seen can be defined compactly in terms of fold:

```
# let listSum l =
      fold (fun a b -> a + b) l 0;;
val listSum : int list -> int = <fun>

# let length l =
      fold (fun a b -> b + 1) l 0;;
val length : 'a list -> int = <fun>

# let filter p l =
     fold (fun a b -> if p a then (a::b) else b) l [];;
```

Using fold

```
# (* List of numbers from m to n, as before *)
  let rec fromTo m n =
          if n < m then []
          else m :: fromTo (m+1) n;;
val fromTo : int -> int -> int list = <fun>

# let fact n =
      fold (fun a b -> a * b) (fromTo 1 n) 1;;
val fact : int -> int = <fun>
```

Forms of fold

OCaml List module actually provides two folding functions

```
List.fold_left
      : (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a
List.fold_right
      : (’a -> ’b -> ’b) -> ’a list -> ’b -> ’b
```

The one we’re calling fold is List.fold_right.

List.fold_left performs the same basic operation but takes its arguments in a different order.

The unit type

OCaml provides another *built-in type* called unit, with just one inhabitant, written ().

```
# let x = ();;
val x : unit = ()

# let f () = 23 + 34;;
val f : unit -> int = <fun>

# f ();;
- : int = 57
```

Why is this useful?

Every function in a functional language must *return a value* . Unit is commonly used as the value pf a procedure that computes by side-effect.

Use of unit

A function from unit to ’a is usually a *delayed computation* of type ’a. For example,

```
# let f () = <long and complex calculation>;;
val f : unit -> int = <fun>
```

... the *long and complex calculation* is just boxed up in a *closure* that we can save for later (by binding it to a variable, e.g.). When we actually need the result, we apply f to () and the calculation actually happens:

```
# f ();;
- : int = 57
```

Thunks

A function accepting *a unit argument* is often called a *thunk*, which is widely used in functional programming.

Suppose we are writing a function where we need to make sure that some "*finalization code*" gets executed, even if an exception is raised.

Thunks

```
# let read file =
    let chan = open_in file in
    try
        let nbytes = in_channel_length chan in
        let string = String.create nbytes in
        really_input chan string 0 nbytes;
        close_in chan;
        string
      with exn ->
        (* finalize channel *)
        close_in chan;
        (* re-raise exception *)
        raise exn;;
```

Thunks

```
# let read file =
    let chan = open_in file in
    let finalize () = close_in chan in
    try
        let nbytes = in_channel_length chan in
        let string = String.create nbytes in
        really_input chan string 0 nbytes;
        finalize ();
        string
      with exn ->
        (* finalize channel *)
         finalize ();
        (* re-raise exception *)
        raise exn;;
```

Thunks: go further...

```
# let unwind_protect body finalize =
    try
            let res = body() in
            finalize();
            res
     with exn ->
            finalize();
            raise exn;;
# let read file =
      let chan = open_in file in
      unwind_protect
            (fun () ->
                    let nbytes = in_channel_length chan in
                    let string = String.create nbytes in
                    really_input chan string 0 nbytes;
                    string)
            (fun () -> close_in chan);;
```

Reference Cell

```
# let fact n =
     let result = ref 1 in
        for i = 2 to n do
          result := i * !result
        done;
     !result;;
val fact : int -> int = <fun>

# fact 5;;
 - : int = 120
```

updatable memory cells, called *references*: ref init returns a new cell with initial contents init, !cell returns the current contents of cell, and cell := v writes the value v into cell.

The rest of OCaml

We’ve seen only a small part of the OCaml language. Some other highlights:

- advanced module system
- imperative features (ref cells, arrays, etc.); the “mostly functional” programming style
- objects and classes

Closing comments on OCaml

Some common strong points of OCaml, Java, C#, etc.
- strong, static typing (no core dumps!)
- garbage collection (no manual memory management!!)

Some advantages of Ocaml compared to Java, etc.
- excellent implementation (fast, portable, etc.)
- powerful module system
- streamlined support for higher-order programming
- sophisticated pattern matching (no “visitor patterns”)
- parametric polymorphism (Java and C# are getting this “soon”)

Some disadvantages:
- smaller developer community
- smaller collection of libraries
- object system somewhat clunky

Performance

It's said that OCaml is fast, way faster than Haskell

- OCaml performed very well in the previous ICFP contests

The reason for OCaml's excellent performance:

- strict evaluation
- the compiler
- mutable data structures

Or as some would say *trading elegance for efficiency*.

Input & Output

Standard built-in I/O functions

I/O Library

Two data types:

- *in_channel*: where characters can be *read* from
- *out_channel*: where characters can be *written* to

There are 3 channels open at program startup:

val stdin : in_channel

val stdout : out_channel

val stderr : out_channel

File opening & closing

Two functions to open an *output file*:

- *open_out*: open a file for writing *text* data
  - val open_out: string -> out_channel
- *open_out_bin*: open a file for writing *binary* data
  - val open_out_bin: string -> out_channel

Two functions to open an *input file*:

- *open_in*: open a file for reading *text* data
  - val open_in: string -> in_channel
- *open_in_bin*: open a file for reading *binary* data
  - val open_in_bin: string -> in_channel

File opening & closing

Two *sophisticated opening functions*, requires an argument of type open_flag:

- *open_in_gen:*

  val *open_in_gen*: open_flag list -> int -> string -> in_channel

- *open_out_gen*:

  val *open_out_gen*: open_flag list -> int -> string -> out_channel

```
type open_flag =
        Open_rdonly | Open_wronly | Open_append
    | Open_creat | Open_ trunc | Open_excl
    | Open_binary | Open_text | Open_nonblock
```

File opening & closing

Functions to *close the channel*:

- *close_in*:

  val close_in: out_channel -> unit

- *close_out*:

  val close_out : out_channel -> unit

If you forget to close a file. The *garbage collector* will eventually close it for you.

However, a good practice is to close the channel manually once you are done with it.

Writing/reading values on a channel

val output_char: out_channel -> char -> unit (write a single character)

val output_string: out_channel -> string -> unit (write all the characters in a string)

val output : out_channel -> string -> int -> int -> unit (write part of a string, offset and length)

val input_char: in_channel -> char (read a single character)

val input_line: in_channel -> string (read an entire line)

val input : in_channel -> string -> int -> int -> int (raise the *exception* End_of_file if the end of the file is reached before the entire value could be read)

Writing/reading values on a channel

Functions for *passing arbitrary OCaml values on a channel* opened in binary mode:

- Read/write a single byte value

val output_byte: out_channel -> int -> unit

val input_byte: in_channel -> int

- Read/write a single integer value

val output_binary_int: out_channel -> int -> unit

val input_binary_int: in_channel -> int

- Read/write arbitrary OCaml values, *unsafe!*

val output_value: out_channel -> ‘ -> unit

val input_value: in_channel -> ‘   (returns a value of arbitrary type ‘ and Ocaml make no effort to check the type)

Channel manipulation

Functions to *modify the position* in a file:

- change the file position

  ```
  val  seek_out: out_channel -> int -> unit
  val  seek_in: in_channel -> int -> unit
  ```

- return the current position in the file

  ```
  val  pos_out: out_channel -> int
  val  pos_in: in_channel -> int
  ```

- return the total number of characters in the file

  ```
  val  pos_out: out_channel -> int
  val  pos_in: in_channel -> int
  ```

Files

Compilation units

Programs

File vs ADT

Modules for *data hiding* & *encapsulation*, including

1. Interface/Signature : *.mli
2. Implementation : *.ml

Ocaml provides *module system* that makes it easy to use the concepts of encapsulation & data hiding

- Every program file acts as *an abstract module*, and called a compilation unit

Files: Signatures

- A Signature contains
  - Type definitions
  - Function declarations

  for the visible types and methods in the module

- A module signature usually has three parts:
  - Data types used by the module
  - Exception used by the module
  - Method type declarations for all the externally visible methods defined by the module

- Type declaration in a signature can be
  - *Abstract* (declaring a type without giving the type definition)
  - *Transparent* (declaring a type including the type definition)

Files: Implementation

Module Implementation is defined in a .ml file with the same base name as the signature file, and consists of

- Data types used by the module.
- Exception used by the module.
- Method definitions

Source file is stored in a file with .ml (mli) suffix, and *;; terminators are not necessary*.

Building a program

Once a *compilation unit* is defined, the types and methods can be used by other files by prefixing the names (of the methods) with the *capitalized* file name

Compiling a program

Using ocamlc, whose usage is much like cc, to compile, and produce files with suffix *.cmo (byte-code version)

```
% ocamlc –c filename.ml
% ocamlc –c filename.mli
```

Another compiler: ocamlopt => *.cmx (native machine code, roughly 3 times faster)

The *.cmo files can be linked by

```
% ocamlc –o  outputfile *.cmo  *.cmo     (default a.out)
```

Order dependent !!

Using ocamldebug, whose usage is much like GNU gdb, to debug a program complied with ocamlc (back command *will go back one instruction)*

```
% ocamlc  –c  –g   …..
% ocamlc  –o  –g   …..
```

Expose a namespace

Using statement

open module_name

to *open a module interface*, which allow *the use of unqualified names for* types, exceptions, and methods.

- Using the full name module_name. method_name to refer is okay, but tedious

Note: multiple opened modules will define the same name.

- The *last* module with open statement will determine the value of the symbol.
- Fully qualified names can be used to access values that may have been hidden by open statement

Utilities
in
OCaml System

Where are we going?

Overall goal:

- we want to turn strings of characters – *code* – into *computer instructions*

Easiest to break this down into phases:

- First, turn strings into *abstract syntax trees* (ASTs) – this is **parsing**
- Next, turn abstract syntax trees into executable instructions – **compiling** or **interpreting**

Lexing and Parsing

Strings are converted into ASTs in two phases:

Lexing Convert strings (streams of characters) into lists (or streams) of *tokens*,, representing words in the language (*lexical analysis*)

Parsing Convert lists of tokens into abstract syntax trees (*syntactic analysis*)

Lexing

With lexing, we break sequences of characters into different syntactic categories, called *tokens*.

As an example, we could break:

asd 123 jkl 3.14

into this:

[String “asd”, Int 123; String “jkl”; Float 3.14]

Lexing Strategy

Our strategy will be to leverage regular expressions and finite automata to recognize tokens:

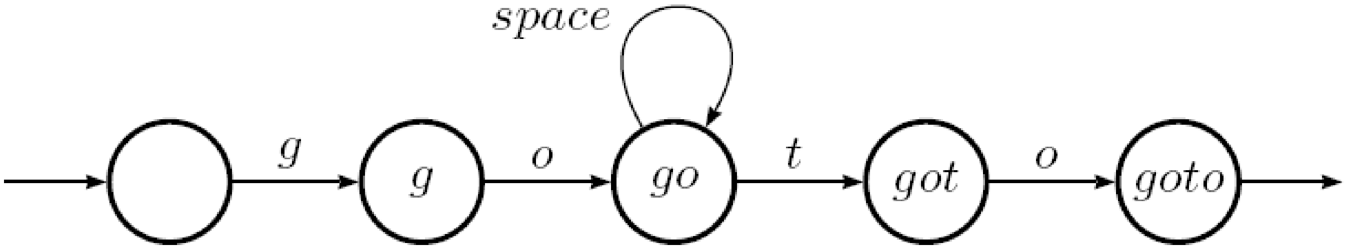
- each syntactic category will be described by a *regular expression* (with some extended syntax)
- words will be recognized by an encoding of a corresponding *finite state machine*

However, this still leaves us with a problem. How do we pull multiple words out of a string, instead of just recognizing a single word?

Lexing : Multiple tokens

To solve this, we will modify the behavior of the DFA.

- if we find a character where there is no transition from the current state, stop processing the string
- if we are in an accepting state, return the token corresponding to what we found as well as the remainder of the string
- now, use iterator or recursion to keep pulling out more tokens
- if we were not in an accepting state, fail – invalid syntax



Lexing Options

We could write a lexer *by writing regular expressions*, and then translating these *by hand* into a DFA.

sounds tedious and repetitive – perfect for a computer!

Can we write a program that takes regular expressions and generates automata for us?

Someone already did – Lex!

- GNU version of this is flex
- *OCaml version* of this is *ocamllex*

How does it work?

We need a few *core items* to get this working:

- Some way to identify the input string – we'll call this the *lexing buffer*
- A set of *regular expressions* that correspond to tokens in our language
- A corresponding *set of actions* to take when tokens are matched

The lexer can then take the regular expressions to build state machines, which are then used to process the lexing buffer.

- If we reach an *accept state* and can *take no further transitions*, we can apply the actions.

Syntax of lexer definitions

```
(*head sections*)
{ header }
(*definition sections*)
let ident = regexp …
(*rule sections*)
rule entrypoint [$arg_1$… $arg_n$] =
            parse regexp { action }
                 |  …
                 |  regexp { action }
and entrypoint [$arg_1$… $arg_n$] =
            parse …
and …
(*rule sections*)
{ trailer }
```

Comments are delimited by (* and *), as in OCaml.

The parse keyword can be replaced by the shortest keyword

Entry points

The names of the *entry points* must be *valid identifiers* for OCaml values (starting with *a lowercase letter*).

Each entry point becomes an OCaml function that takes *n+1* arguments

- arguments arg_1… arg_n must be valid identifiers for Ocaml
- the extra implicit *last* argument being of type Lexing.lexbuf, Characters are read from the Lexing.lexbuf argument and matched against the regular expressions provided in the rules, until a prefix of the input matches one of the rules.
- the corresponding action is then evaluated and returned as the result of the function.

Regular Expressions in ocamllex

The regular expression format is similar to what we’ve seen so far, but still slightly different.

- **‘ *regular-char* | *escape-sequence* ’** A character constant, with the same syntax as OCaml character constants. Match the denoted character.
- **_** (underscore) Match any character.
- **eof** Match the end of the lexer input.
- **“ { *string-character* } ”** A string constant, with the same syntax as OCaml string constants. Match the corresponding sequence of characters.
- **[*character-set*]** Match any single character belonging to the given character set. Valid character sets are: single character constants ' c '; ranges of characters ' c_1 ' - ' c_2 ' (all characters between c_1 and c_2, inclusive); and the union of two or more character sets, denoted by concatenation.
- **[^ *character-set*]** Match any single character not belonging to the given character set.

Regular Expressions in ocamllex

- ***regexp*$_1$ # *regexp*$_2$** (difference of character sets) Regular expressions *regexp*$_1$ and *regexp*$_2$ must be character sets defined with […] (or a a single character expression or underscore _). Match the difference of the two specified character sets.
- ***regexp* ***(repetition) Match the concatenation of zero or more strings that match *regexp*.
- ***regexp* +**(strict repetition) Match the concatenation of one or more strings that match *regexp*.
- ***regexp* ?**(option) Match the empty string, or a string matching *regexp*.

Regular Expressions in ocamllex

- ***regexp***$_1$ **|** ***regexp***$_2$ (alternative) Match any string that matches *regexp*$_1$ or *regexp*$_2$
- ***regexp***$_1$ ***regexp***$_2$ (concatenation) Match the concatenation of two strings, the first matching *regexp*$_1$, the second matching *regexp*$_2$.
- **(** ***regexp*** **)** Match the same strings as *regexp*.
- ***ident*** Reference the regular expression bound to *ident* by an earlier let *ident* = *regexp* definition.
- ***regexp*** **as** ***ident*** Bind the substring matched by *regexp* to identifier *ident*.

Actions

Can be arbitrary OCaml expressions. They are evaluated in a context where the identifiers defined by using the *as construct* are bound to subparts of the matched string.

Additionally, lexbuf is bound to the current lexer buffer. Some typical uses for lexbuf:

- **Lexing.lexeme lexbuf** Return the matched string.
- **Lexing.lexeme_char lexbuf *n*** Return the n^{th} character in the matched string. The first character corresponds to $n = 0$.
- **Lexing.lexeme_start lexbuf** Return the absolute position in the input text of the beginning of the matched string (i.e. the offset of the first character of the matched string). The first character read from the input text has offset 0.
- **Lexing.lexeme_end lexbuf** Return the absolute position in the input text of the end of the matched string (i.e. the offset of the first character after the matched string).
- ***entrypoint* [exp_1 … exp_n] lexbuf** Recursively call the lexer on the given entry point

Header and trailer

Can be arbitrary OCaml text enclosed in curly braces.

- Either or both can be omitted. If present, the header text is copied as is at the beginning of the output file and the trailer text at the end.
- Typically, the header section contains the *open directives* required by the actions, and possibly some auxiliary functions used in the actions.

Sample Lexer

```
rule main = parse
   |  [’0’-’9’]+   { print_string “Int\n"}
   |  [’0’-’9’]+’.’[’0’-’9’]+  { print_string "Float\n"}
   |  [’a’-’z’]+  { print_string "String\n"}
   |  _  { main lexbuf }
{
  let newlexbuf = (Lexing.from_channel stdin)  in
      print_string "Ready to lex.\n";
  main newlexbuf
}
```

Mechanics of Using ocamllex

Lexer definitions using ocamllex are written in a file with a .mll extension.

- including the regular expressions, with associated actions for each.

OCaml code for the lexer is generated with

ocamllex *lexer*.mll

this generates the code for the lexer in file file.ml

- This file defines one lexing function per entry point in the lexer definition

Options for ocamllex

The following command-line options are recognized by *ocamllex.*

- **ml** Output code that does not use OCaml’s built-in automata interpreter. Instead, the automaton is encoded by OCaml functions. This option mainly is useful for debugging ocamllex, using it for production lexers is not recommended.
- **o *output-file*** Specify the name of the output file produced by ocamllex. The default is the input file name with its extension replaced by .ml.
- **q** Quiet mode Ocamllex normally outputs informational messages to standard output. They are suppressed if option -q is used.
- **v** or **–version** Print version string and exit.
- **Vnum** Print short version number and exit.
- **help** or **– help** Display a short usage summary and exit.

Parsing

Convert lists of tokens into abstract syntax trees

Someone already did – Yacc!

- GNU: bison
- Ocaml: **ocamlyacc**

Yacc

provides a general tool for describing the input to a *computer program*.

- The Yacc user specifies the *structures* of his input, together with *code* to be invoked as each such structure is recognized.
- Yacc turns such a specification into a *subroutine* that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

ocamlyacc Command

Produces a parser from a context-free grammar specification with attached semantic actions, in the style of yacc.

Executing

```
ocamlyacc options grammar.mly
```

produces OCaml code for a parser in the file grammar.ml, and its interface in file grammar.mli.

- The generated module defines one parsing function per entry point in the grammar. These functions have the same names as the entry points.
- Parsing functions take as arguments a lexical analyzer (a function from lexer buffers to tokens) and a lexer buffer, and return the semantic attribute of the corresponding entry point.

Options for ocamlyacc

-b*prefix*	Name the output files *prefix*.ml, *prefix*.mli, *prefix*.output, instead of the default naming convention.
-q	This option has no effect.
-v	Generate a description of the parsing tables and a report on conflicts resulting from ambiguities in the grammar. The description is put in file grammar.output.
-version	Print version string and exit.
-vnum	Print short version number and exit.
-	Read the grammar specification from standard input. The default output file names are stdin.ml and stdin.mli.
-- *file*	Process file as the grammar specification, even if its name starts with a dash (-) character. This option must be the last on the command line.

Syntax of grammar definitions

%{

header

%}

declarations

%%

rules

%%

trailer

Comments are enclosed between /* and */ (as in C) in the "declarations" and "rules" sections, and between (* and *) (as in OCaml) in the "header" and "trailer" sections.

header and trailer

OCaml code that is copied as is into file grammar.ml.

- Both sections are optional.
- The header goes at the beginning of the output file; it usually contains open directives and auxiliary functions required by the semantic actions of the rules.
- The trailer goes at the end of the output file.

Declarations

given one per line. They all start with a % sign.

%token *constr* … *constr*

%token < *typexpr* > *constr* …

Declare the given symbols *constr* … *constr* as tokens (terminal symbols).

%start *symbol* … *symbol*

Declare the given symbols as entry points for the grammar. For each entry point, a parsing function with the same name is defined in the output module

%type < *typexpr* > *symbol* … *symbol*

Specify the type of the semantic attributes for the given symbols. This is mandatory for start symbols only

%left *symbol* … *symbol*

%right *symbol* … *symbol*

%nonassoc *symbol* … *symbol*

Rules

The syntax for rules is as usual:

nonterminal :
    symbol … *symbol* { *semantic-action* }
  | …
  | *symbol* … *symbol* { *semantic-action* }
;

Rules can also contain the %prec *symbol* directive in the right-hand side part, to override the default precedence and associativity of the rule with the precedence and associativity of the given symbol.

Semantic actions are *arbitrary OCaml expressions*, that are evaluated to produce the semantic attribute attached to the defined nonterminal.

The semantic actions can access the semantic attributes of the symbols in the right-hand side of the rule with the $ notation:

- $1 is the attribute for the first (leftmost) symbol, $2 is the attribute for the second symbol, etc.

Utilities
in Environment

make

one critical utility in the Unix/Linux-like environment

- 自动管理、检查文件之间的依赖关系
- 自动判断哪些文件要重新编译, 调用外部程序进行处理
  - 根据文件的修改时间
- 常用于编译源文件生成目标文件, 将目标文件链接成可执行文件或库

makefile

- 用文件 ” makefile” 或 ” Makefile” 描述依赖和动作，动作由shell 执行
- 命令make解释”makefile”

Makefile for hello

e.g., GNU make

```
hello: hello.c
	gcc hello.c -o hello
```

```
$make
gcc hello.c –o hello
```

目标和依赖

makefile 由如下的一系列规则组成

```
target1 target2  target3 :  prerequisite1, prerequisite2
      command1
      command2
```

目标和依赖说明

- 目标(target)：要做的事情，要生成的文件
- 倚赖(prerequisite)：在生成目标前，其所有倚赖必须存在
- 命令(command)：根据依赖生成目标的shell 命令.命令前必须是缩进(tab)
- makefile 中的第一个规则称为缺省目标(goal)

工作过程

- 若在命令行给出了目标，则make 找到该目标的规则；否则执行缺省目标
- 对于每个规则，首先查看所有的依赖和目标
  - 若某个依赖有规则，则首先处理该依赖的规则
  - 若某个依赖的时间比目标新，则执行命令更新目标
  - 命令由shell 执行，若执行错误，则中止处理

规则

- 显式规则(explicit rule): makefile 中显式声明的规则，如 vpath.o variable.o: make.h config.h dep.h
- 隐式规则(implicit rule): make 内置的模式规则或后缀规则
  - 在GNU make 中，后缀规则可被模式规则代替
- 模式规则(pattern rule): 用通配符取代显式的文件名，跟Bourne sh 相同， 如

  ~ * ? [...] [^...]

变量

在makefile 中可以定义变量: Name = Value

随后通过$(Name) 或 ${Name} 访问

make 的自动变量

$@	目标文件名
$%	档案文件(库) 的成员
$<	第一个依赖文件的文件名
$?	所有比目标文件新的倚赖文件名列表, 以空格分隔
$^	所有依赖文件名列表, 以空格分隔
$+	和$^ 类似, 包含重复文件名
$*	目标文件名去除后缀后的部分

An Implementation for Arithmetic Expression

Structure of arith

main.ml drives the whole process

- Scan tokes (lexer.mll)
- Parse terms (parser.mly)
- Evaluate each terms (eval in core.ml)
- Print the values (printtm in syntax.ml)

syntax.ml defines the terms

Makefile

```
#
# Rules for compiling and linking the typechecker/evaluator
#
# Type
#   make         to rebuild the executable file f
#   make windows to rebuild the executable file f.exe
#   make test    to rebuild the executable and run it on input file test.f
#   make clean   to remove all intermediate and temporary files
#   make depend  to rebuild the intermodule dependency graph that is used
#                by make to determine which order to schedule
#                      compilations.  You should not need to do this unless
#                you add new modules or new dependencies between
#                existing modules.  (The graph is stored in the file
#                .depend)

# These are the object files needed to rebuild the main executable file
#
OBJS = support.cmo syntax.cmo core.cmo parser.cmo lexer.cmo main.cmo

# Files that need to be generated from other files
DEPEND += lexer.ml parser.ml
```

Syntax.ml

```
type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

info: a data type recording the position of the term in the source file

eval in core.ml

```
let rec eval t =
  try let t’ = eval1 t
      in eval t’
  with NoRuleApplies → t
```

eval1: perform a single step reduction

Commands

- Each line of the source file is parsed *as a command*
  - type command = | Eval of info * term
  - New commands will be added later

- Main routine for each file

```
 let process_file f  =
      alreadyImported := f :: !alreadyImported;
      let cmds = parseFile f in
      let g  c =
          open_hvbox 0;
          let results = process_command  c in
    print_flush();
    results
 in
   List.iter g  cmds
```

Exercise arith.simple_use

- Using arith to write the following equation
  - – Return five if two is not zero, otherwise return nine
  - – Hint: read the code in parser.mly

Homework

- Please get familiar with OCaml and its utilities
- Please download the implementation package of the TAPL, and digest the source codes in archives of *arith*, *tyarith*, *untype*.
- On this basis, please give your implementation for Chap. 4
  - Submit your code as a compressed file with one of the above names
  - Your submission should contain file test.f which contains exactly the expressions to be tested
  - TA will perform the following two commands to verify your submission:
    - make
    - ./f test.f