CS131, Spring19 – Discussion 1B

Week 3 (04/19/19)

Administrations

1. TA: Wenhao Zhang (wenhaoz@cs.ucla.edu)
2. Office hour:
    - Time: 9:30am ~ 11:30am, Monday
    - Location: 3rd floor, common area in Eng VI
3. Class Announcements
    - All TAs slides are under **resources tab** on Piazza
    - HW2 dues on **04/21** (This Sunday)
    - Midterm: Thursday, 05/02.
    - Midterm review in next discussion
    - Please dont copy & paste code from stackoverflow or other sources.
4. Today's agenda
    - Recap of last week
    - More on HW2
    - Java intro

Quick recap from last week

- Higher order function && Currying
- Q: Use List.fold_left to write a function which concatenates a list of strings.
    - Recall from last week
    - `let sum = List.fold_left (+) 0;;`
    - Answer: `let f = List.fold_left (^) "";;`
- Q: what's the type of this one?
    - `let f f = f 1 1;;`
    - `let f = fun f -> f 1 1;;`
    - 'let f1 = fun f -> f 1 1;;'
- Grammar recap
    - HW2 Grammar

```
(Expr,
    function
        | Expr -> [[N Term];
                [N Term; N Binop; N Expr]]
        | Term -> ...
)
```

- Left-most Derivation "3" "+" "4"

```
Expr → Term Binop Expr | Term
Term → Num | Lvalue | Incrop Lvalue | Lvalue Incrop | "(" Expr ")"
Lvalue → "$" Expr
```

```
Incrop → "++" | "--"
Binop → "+" | "-"
Num → "0" | "1" | ... | "9"
```

Leftmost derivation always expands the leftmost nonterminal next.

```
(start)                     Expr
Expr → Term Binop Expr      Term Binop Expr
Term → Num                  Num Binop Expr
Num → "3"                   "3" Binop Expr
Binop → "+"                 "3" "+" Expr
Expr → Term                 "3" "+" Term
Term → Num                  "3" "+" Num
Num → "4"                   "3" "+" "4"
```

- **Frag** This is simply a list of terminal symbols. This is equivalent to a program that you are trying to parse.
- **Prefix** In the above left most derivation, "3" "+" "4" is a prefix.
- **Suffix** In the above left most derivation, [] is the suffix since every terminal in frag has been matched.

More on HW2

Option

Options are are an Ocaml standard type that can be either None (undefined) or Some x where x can be any value. Options are widely used in Ocaml to represent undefined values (a little like NULL in C, but in a type and memory safe way).

```
type `a option =
| None
| Some of `a

(*This expression has the following type*)
(*int option = Some 1*)
Some 1;;
```

Acceptor:

a function whose argument is a fragment frag. If the fragment is not acceptable, it returns None; otherwise it returns Some x for some value x.

- Simple Acceptor which accepts all

  ```
  let accept_all suffix = Some suffix;;
  ```

- Acceptor that only accepts suffixes that are not empty

  ```
  let accept_nonempty  = function
   | [] -> None
   | s -> Some s;;
  ```

Basic idea is that an acceptor is simply a function, it has no implicit meaning.

Matcher

A curried function with two arguments, 1) an acceptor accept and 2) a fragment frag. A matcher matches a prefix p of frag such that accept (when passed the corresponding suffix) accepts the corresponding suffix (i.e., the suffix of frag that remains after p is removed). If there is such a match, the matcher returns whatever accept returns; otherwise it returns None.

How does matcher work exactly?

1. find the matching prefix using grammar derivation
2. if no prefix is found: return None.
3. else call acceptor on suffix return whatever acceptor returns.

For example, let's say we have a fragment `["3", "+", "4", "-"]`, and you want to parse it using the given grammar in hw2 specs. Your matcher finds two possible **prefixes**, ["3", "+", "4"] and ["3"]. This can be easily checked using `Expr->[N Term; N Binop; N Expr] or Expr -> [N Term]` in `awkish_grammar`. The corresponding **suffixes** of `["3", "+", "4"] and ["3"]` are `["-"] and`["+", "4", "-"]':

1) If the acceptor only accepts suffix `["-"]`, then your matcher found a good prefix (i.e. `["3", "+", "4"]`), and will **return whatever the acceptor returns**.

2) If the acceptor only accepts **empty suffix**, then you run out of options because the acceptor rejects the two suffixes metioned above. **In this case, your matcher fails to find a matching prefix, and returns None**.

The next task for you is to write a curried func `make_parser gram` which returns a parser for the grammar gram. When applied to a fragment frag, the parser returns an optional parse tree. If frag cannot be parsed entirely (that is, from beginning to end), the parser returns None.

```
let test2 =
  ((make_matcher awkish_grammar accept_all ["9"; "+"; "$"; "1"; "+"])
   = Some ["+"]

let test3 =
  ((make_matcher awkish_grammar accept_empty_suffix ["9"; "+"; "$"; "1"; "+"])
   = None)
```

Let's write some simple matchers

Q1. Write a matcher that matches empty prefix

```
(*Solution to Q1*)
let match_empty accept frag = accept frag
```

```
(*type of this matcher*)
val match_empty : ('a -> 'b) -> 'a -> 'b = <fun>

(*example of using match_empty*)
match_empty accept_all [1;2;3];;    (*=> Some [1;2;3]*)
```

Q2. Single element matcher

This `make_match_start` function takes in 3 arguments: v : 'a, frag : 'a list, and accept function. **make_matcher pattern returns a matcher for the pattern** This matcher tries to match v to the first element of frag.

```
(*Solution to Q2*)
let make_match_start v acceptor frag = match frag with
  | [] -> None
  | f::r -> match f with
            | v -> acceptor r;;

(*type of this `matcher function`*)
val make_match_start : 'a -> ('b list -> 'c option) -> 'b list -> 'c option = <fun>

(*example of call make_match_start*)
make_match_start 1 accept_all [1;2;3];;  (*=> Some [2;3]*)

(*match_1 is a matcher (curried function) that only matches frags starts with 1*)
let match_1 = make_match_start 1;;
```

Q3. Append matchers

Suppose we had two matchers, and we wanted to use them both in sequence.

```
(*Question*)
let append_matchers matcher1 matcher2 accept frag = ?

(*Answer*)
let append_matchers matcher1 matcher2  accept frag =
  matcher1 (fun frag1 -> matcher2 accept frag1) frag;;

let matcher_1 = make_match_start 1;;
let matcher_2 = make_match_start 2;;

append_matchers matcher_1 matcher_2  accept_all [1;2;3];;   (*=> Some [3]*)

(*Write a sequence of matchers*)
let make_appended_matchers make_a_matcher ls =
    let rec mams = function
        | [] -> match_empty
        | head::tail -> append_matchers (make_a_matcher head) (mams tail)
in mams ls;;
```

```
(*create a sequence of matchers*)
let accept_empty suffix = match suffix with
  | [] -> Some []
  | _ -> None;;

make_appended_matchers make_match_start [1;2;3] accept_empty [1;2;3];;  (*=> Some []*)

make_appended_matchers make_match_start [1;2;3] accept_empty [1;2;3;4];;    (*=> None*)
```

Q4. Or_matcher

Discuss what his function does

```
let match_nothing frag accept = None

let rec f make_a_matcher = function
  | [] -> match_nothing
  | head::tail ->
      let head_matcher = make_a_matcher head
      and tail_matcher = f make_a_matcher tail
      in fun accept frag ->
       let ormatch = head_matcher accept frag
       in match ormatch with
        | None -> tail_matcher frag accept
        | _ -> ormatch

f make_match_start [1;2;3] accept_all [1;4;5];; (*=> Some [4;5]*)

f make_match_start [1;2;3] accept_all [2;4;5];; (*=> Some [4;5]*)
```

Object-oriented programming (OOP)

Overview

- Objects are the first-class citizens.
- Objects encapsulate related methods and fields
- Example languages e.g. Java, C++, C#, Python, PHP, JavaScript, Ruby, etc

Class

Class is a template for an object. Object is an instance of a class. **All objects created using the same class will have the same methods/fields.**

What are the benefits of OOP?

- Modularity
  - Splitting code into objects can help keep different parts of code separated
- Information-hiding
  - Objects should only interact by using each other's public methods
- Code reuse
  - Objects easy to re-use in other programs
- Pluggability and debugging ease
  - You can easily replace a buggy object with a working one if necessary

Alan Kay's definition of OOP

- Everything is an object
- Objects communicate by sending/receiving messages
- Objects have their own memory
- Every object is an instance of some class
- All objects of a specific type can receive the same messages

Note: Some of these do not apply to all of the modern OOP languages!

Java intro

- We will be using Java 11 for this class.
- We recommend you to use an IDE like Eclipse, Netbeans, IntelliJ.
  - This will give you autocomplete, debugging, syntax highlighting and other features to make your life easier
- Other option would be a text editor (e.g. Emacs, sublime, Vim) + the terminal
  - Compile with: javac fileName.java
  - Run with: java fileName

Hello World

```java
/*HelloWorld.java*/

public class HelloWorld {
	public static void main(String[] args) {
		System.out.println("Hello, World");
	}
}
```

Java files

- MyClass.java
  - Code for MyClass
- MyClass.class
  - Bytecode for MyClass (Compiled using `javac MyClass.java`)
- Foo.jar
  - Java Archive file; ZIP archive
  - Could contain dependent code files or other resources
  - In HW3, you are provided a jar file containing the necessary code files

Inheritance

```
class Shape {
      void draw() { /* do nothing */ }
}
class Rectangle extends Shape {
      void draw() { /* draw a rectangle */ }
}
class Circle extends Shape {
      void draw() { /* draw a circle */ }
}
class Triangle extends Shape {
      void draw() { /* draw a triangle */ }
}
```

```
Triangle a = new Triangle(); a.draw(); /* draws a triangle */
Shape b = a;
b.draw(); /* draws a triangle */
b = new Circle();
b.draw(); /* draws a circle */
```

Interface

- Defines what a class must be able to do, not how to do it
- Interface can not be instantiated, must create a class that implements that interface
- One class can implement multiple interfaces

```
interface Vehicle {
      public int currentSpeed;
```

```
      public void increaseSpeed();
      public void decreaseSpeed();
      public void turnLeft();
      public void turnRight();
}

class Car implements Vehicle {
      public void increaseSpeed() {
             pressGasPedal();
      }
      public void decreaseSpeed() {
             pressBrakePedal();
      }
... rest of the implementations ...
}
```

Abstract Classes

- Abstract classes are a combination of a class and an interface
  - Can't create an object using an abstract class
  - Can define some parts of the class, while leaving other implementations for children
- Classes can extend only one abstract or normal class

```
abstract class Shape {
     abstract void draw();
     void setColor() { /* set color */ }
}
```