CS 131
PROGRAMMING LANGUAGES
(WEEK 2)

UCLA WINTER 2019

TA: SHRUTI SHARAN

DISCUSSION SECTION: 1D

ADMINISTRATIVE INTRODUCTION

TA: Shruti Sharan

Email: shruti5596@g.ucla.edu
(Will reply by EOD)

Office Hours: Mondays 1.30PM – 3.30PM
Location: Eng. VI 3rd Floor

Discussion Section: Friday 4.00-5.50PM
Location: 2214 Public Affairs

TODAY'S AGENDA

1. RECAP OF LAST WEEK
2. HIGHER ORDER FUNCTIONS AND CURRYING
3. GRAMMAR AND DERIVATION
4. DISCUSSION OF HW2

HOW TO FIND THESE SLIDES

- Piazza -> CS 131 -> Resources -> Discussion 1D

HOMEWORK ANNOUNCEMENTS

- HW1 **was** due on 16/01. (Please submit it soon if you haven't already.)
- HW2 is posted. Due on Tuesday, 29/01 11:55pm.
- All homework should be submitted to CCLE.
- Some homework will have automated grading scripts
  - Make sure code compiles
  - Make sure that you follow the function signatures
  - Follow all the instructions and specifications

RECAP : FUNCTIONS

- Define using **let** keyword.
- Takes only one argument.
- Evaluation:
  - Prints inferred types (type checking)
  - Compiles -> executes -> prints result
- ALL TYPES are allowed as arguments in functions.

POLYMORPHIC FUNCTIONS

- Polymorphic functions can have parameters of different types.
- OCaml's type inference determines that an expression is valid for any type, it is automatically made polymorphic, parameterized by type variables.
- Type variables are lowercase identifiers preceded by a single quote ', normally 'a, 'b, 'c and so on.

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

```
# id 1;;
- : int = 1
# id "OCaml";;
- : string = "OCaml"
# id 4.5;;
- : float = 4.5
```

LAMBDA FUNCTIONS

- Lambda functions (aka Anonymous functions) are not bound to any name
- Useful when using a function as a function parameter
  - Very common in functional programming!
  - “Higher-order function”

```
# (fun x -> x*x) 5;;
- : int = 25
```

HIGHER ORDER FUNCTIONS

- This means that we can pass functions around as arguments to other functions.
- We can store functions in data structures
- We can return functions as a result from other functions.

```
# let double x = 2 * x ;;
val double : int -> int = <fun>
# let quad x = double ( double x);;
val quad : int -> int = <fun>
```

```
# let twice f x = f(f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
# let quad x = twice double x;;
val quad : int -> int = <fun>
```

- The function twice is higher-order:
  - its input f is a function
  - Since OCaml functions really take only a single argument—its output is fun x -> f (f x) so twice returns a function hence is also higher-order in that way.

CURRYING

- A function with multiple parameters is really just syntactic sugar for a function that is passed as a tuple as an argument.
- Break a function with arguments into multiple functions each taking only a single argument. (A chain of functions gets created).
- One can create functions at run time. Computed values of functions is remembered in bytecode.
- Since only one argument is accepted, we can pass arguments in a single structure: **Tuples.**
  - N-tuple : ordered sequence of n values separated by commas: $(e_1, e_2, \dots \dots \dots\ e_n)$

CURRYING

- We can parenthesize the term as (multiply 5) (6), because application is left-associative.
- In other words, multiply 5 must return a function that can be applied to 6 to obtain the result 30. In fact, multiply 5 returns an anonymous function that multiplies 5 to its argument.

```
# let multiply x y = x * y;;
val multiply : int -> int -> int = <fun>
# multiply 5 6;;
- : int = 30
# (multiply 5) 6;;
- : int = 30
# let multiply5 = multiply 5;;
val multiply5 : int -> int = <fun>
# multiply5 6;;
- : int = 30
```

CURRYING

- The curried declaration above is syntactic sugar for the creation of a **higher-order function.**

```
# multiply 5 6
  = ((function (x : int) -> function (y : int) -> x + y ) 5 ) 6
  = (function (y : int) -> 5 + y) 6
  = 5 * 6
  = 30
```

RECAP : RECURSION

- Must explicitly define using keyword **rec**
- Building block of truly functional solutions

```
# let rec factorial a =
    if a = 1 then 1 else a * factorial (a-1);;
val factorial : int -> int = <fun>
```

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

MUTUAL RECURSION

- Two functions are said to be mutually recursive if the first calls the second, and in turn the second calls the first.

```
# let rec even x =
  match x with
  | 0 -> true
  | x -> odd(x-1)
  and
  odd x =
  match x with
  | 0 -> false
  | x -> even(x-1) ;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
```

```
# even 10;;
- : bool = true
# even 7;;
- : bool = false
# odd 3;;
- : bool = true
```

BUILT IN TYPES : OPTIONS

- Variable with or without a value (safer version of null used in some other languages.
- A value v has type t option if it is either:
  - the value None
  - a value Some v', and v' has type t.

```
# Some 5 ;;
- : int option = Some 5
# Some "Hello" ;;
- : string option = Some "Hello"
# None ;;
- : 'a option = None
```

- Forces you to handle missing values explicitly -> safer code

```
# let rec find_val l v =
  match l with
  | [] -> None
  | h::t when h=v -> Some h
  | h::t -> find_val t v;;
val find_val : 'a list -> 'a -> 'a option = <fun>
```

RECAP: PATTERN MATCHING

- More powerful alternative for conditionals (if - then - else)

```
# let is_zero x = match x with
  | 0 -> true
  | _ -> false ;;
val is_zero : int -> bool = <fun>
# is_zero 0;;
- : bool = true
# is_zero 5;;
- : bool = false
```

Alternative syntax:

```
# let is_zero = function
  | 0 -> true
  | _ -> false ;;
val is_zero : int -> bool = <fun>
```

PATTERN MATCHING: TUPLES

```
# let tuple_matcher = function
  | (1,a) -> a
  | _ -> 0;;
val tuple_matcher : int * int -> int = <fun>
# ;;
# tuple_matcher(1,5);;
- : int = 5
# tuple_matcher(0,5);;
- : int = 0
```

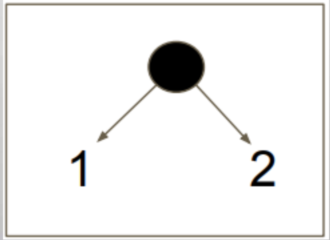
PATTERN MATCHING: TYPES

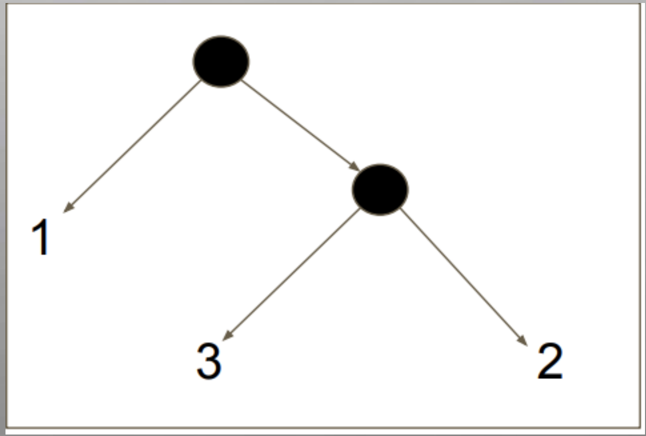
```
# type my_type=
  |A of string
  |B of int ;;
type my_type = A of string | B of int
# let type_matcher = function
  | A a -> "Type A"
  | B b -> "Type B" ;;
val type_matcher : my_type -> string = <fun>
# type_matcher ( A "Some string");;
- : string = "Type A"
# type_matcher ( B 5) ;;
- : string = "Type B"
```

BINARY TREES

- How can we define binary trees in OCaml?
- Data is only present at leaf nodes.

```
# type 'a bintree =
  | L of 'a
  | N of ('a bintree * 'a bintree) ;;
type 'a bintree = L of 'a | N of ('a bintree * 'a bintree)
```





BINARY TREES - BFS

```
type 'a bintree = L of 'a | N of ('a bintree * 'a bintree)
# let rec bfs_help q ret = match q with
  | [] -> ret
  | h ::t -> (match h with
              | L x -> bfs_help t (ret@[x])
              | N (f,s) -> bfs_help ( t @ [f;s]) ret) ;;
val bfs_help : 'a bintree list -> 'a list -> 'a list = <fun>
# let bfs tree = match tree with
  | L x -> [x]
  | N (f,s) -> bfs_help [f;s] [];;
val bfs : 'a bintree -> 'a list = <fun>
```

GRAMMAR : REVIEW

- Symbol
  - Terminal: A symbol which you cannot replace with other symbols
  - Non-terminal: A symbol which you can replace with other symbols
- Rule
  - From a non terminal symbol, derive a list of symbols
- Grammar:
  - A starting symbol, and a set of rules

GRAMMARS - RECAP

- Grammar: A starting symbol, and a set of rules that describe what symbols can be derived from a non-terminal symbol.

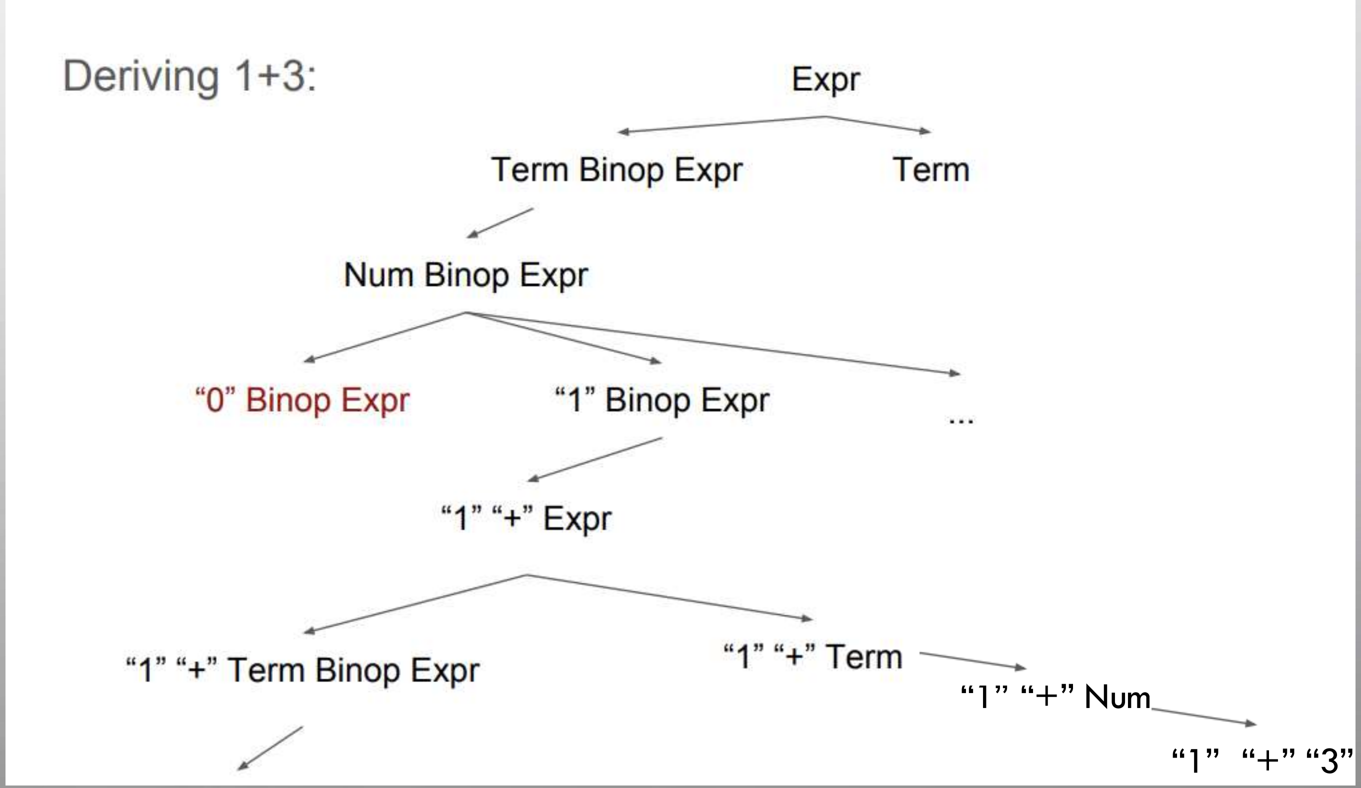
```
let awkish_grammar =
  (Expr,
   function
     | Expr ->
         [[N Term; N Binop; N Expr];
          [N Term]]
     | Term ->
         [[N Num];
          [N Lvalue];
          [N Incrop; N Lvalue];
          [N Lvalue; N Incrop];
          [T"("; N Expr; T")"]]
     | Lvalue ->
         [[T"$"; N Expr]]
     | Incrop ->
         [[T"++"];
          [T"--"]]
     | Binop ->
         [[T"+"];
          [T"-"]]
     | Num ->
         [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
          [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])
```

DERIVATIONS

- Recap of the top-down parsing technique: How to derive 1+3?

Expr → Term Binop Expr
Expr → Term
Term → Num
Term → Lvalue
Term → Incrop Lvalue
Term → Lvalue Incrop
Term → "(" Expr ")"
Lvalue → $ Expr
Incrop → "++"
Incrop → "−−"
Binop → "+"
Binop → "−"
Num → "0"
Num → "1"
Num → "2"
Num → "3"
Num → "4"
Num → "5"
Num → "6"
Num → "7"
Num → "8"
Num → "9"

DERIVATION…. continued



Expr → Term Binop Expr
Expr → Term
Term → Num
Term → Lvalue
Term → Incrop Lvalue
Term → Lvalue Incrop
Term → "(" Expr ")"
Lvalue → $ Expr
Incrop → "++"
Incrop → "−−"
Binop → "+"
Binop → "−"
Num → "0"
Num → "1"
Num → "2"
Num → "3"
Num → "4"
Num → "5"
Num → "6"
Num → "7"
Num → "8"
Num → "9"

HOMEWORK 2

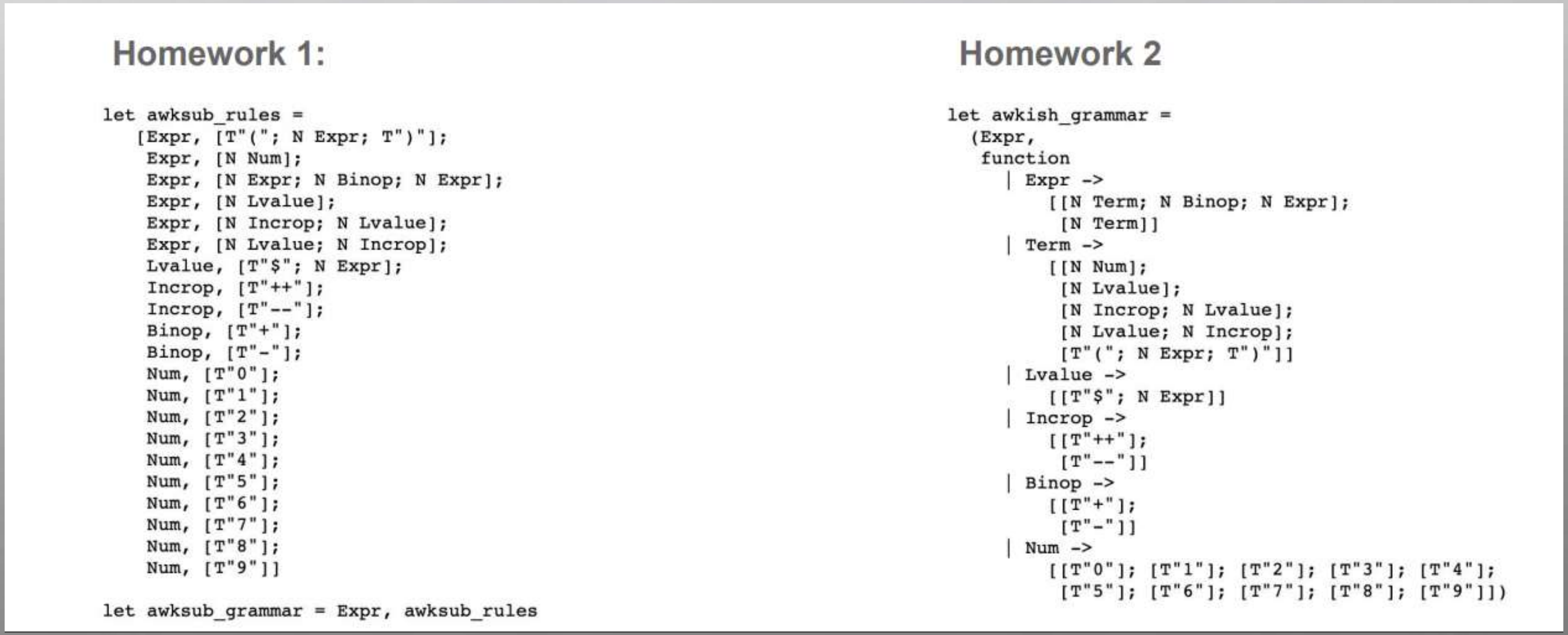
- Posted today.
- Due on 01/29 at 11.55pm.
- More difficult that HW1! (Please start in advance.)
- Check Hint code. Very useful for Homework 2.

HOMEWORK 2

- Go over the first part
- We will continue covering homework 2 next week
- Go through the solution to a similar problem from an earlier year
  - Link provided at the bottom of the homework 2
  - We will discuss this next week too

HOMEWORK 2 - TASK 1 (WARM-UP)

- Our syntax for grammars is slightly different from last week; write a function to convert old syntax to new syntax:



Homework 1:

```
let awksub_rules =
   [Expr, [T"("; N Expr; T")"];
    Expr, [N Num];
    Expr, [N Expr; N Binop; N Expr];
    Expr, [N Lvalue];
    Expr, [N Incrop; N Lvalue];
    Expr, [N Lvalue; N Incrop];
    Lvalue, [T"$"; N Expr];
    Incrop, [T"++"];
    Incrop, [T"--"];
    Binop, [T"+"];
    Binop, [T"-"];
    Num, [T"0"];
    Num, [T"1"];
    Num, [T"2"];
    Num, [T"3"];
    Num, [T"4"];
    Num, [T"5"];
    Num, [T"6"];
    Num, [T"7"];
    Num, [T"8"];
    Num, [T"9"]]

let awksub_grammar = Expr, awksub_rules
```

Homework 2

```
let awkish_grammar =
  (Expr,
   function
     | Expr ->
         [[N Term; N Binop; N Expr];
          [N Term]]
     | Term ->
         [[N Num];
          [N Lvalue];
          [N Incrop; N Lvalue];
          [N Lvalue; N Incrop];
          [T"("; N Expr; T")"]]
     | Lvalue ->
         [[T"$"; N Expr]]
     | Incrop ->
         [[T"++"];
          [T"--"]]
     | Binop ->
         [[T"+"];
          [T"-"]]
     | Num ->
         [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
          [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])
```

HOMEWORK 2 - TASK 1 (WARM-UP)

- Previously, we used a list of tuples for the rules
- Now, a function with pattern matching
- Calling this function with e.g. *Term* returns all the allowed replacement rules for *Term*

```
let awkish_grammar =
  (Expr,
   function
     | Expr ->
         [[N Term; N Binop; N Expr];
          [N Term]]
     | Term ->
         [[N Num];
          [N Lvalue];
          [N Incrop; N Lvalue];
          [N Lvalue; N Incrop];
          [T"("; N Expr; T")"]]
     | Lvalue ->
         [[T"$"; N Expr]]
     | Incrop ->
         [[T"++"];
          [T"--"]]
     | Binop ->
         [[T"+"];
          [T"-"]]
     | Num ->
         [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
          [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])
```

OLD HOMEWORK EXAMPLE

- Build a pattern matcher for genetic sequences
- Genetic sequence consists of letters: A, C, G, T (adenine, thymine, cytosine, and guanine
  - E.g. AGGTCAGTTACAATTGCTT...
  - These are the only allowed symbols in our language
- (If you're interested in genetic sequencing, consider taking CS CM121 - Introduction to Bioinformatics)

PATTERNS

- **Frag** [symbol list]
  - Match a list of symbols, e.g. Frag [C;T;G] matches [C;T;G]
- **Junk** k
  - Matches up to k symbols, e.g. Junk 1 matches [], [A], [C], [T], [G]
- **Or** [pattern list]
  - Matches any pattern in the list, e.g. Or [Frag[C;T]; Frag[A;G]] matches [C;T] and [A;G]
- **List** [pattern list]
  - Matches a concatenation of patterns, e.g. List [Frag[A]; Junk 1; Frag[G]] matches [A;A;G], [A;C;G], [A;T;G], [A;G;G]
- **Closure pattern**
  - Matches a concatenation of patterns, 0 or more times, e.g. Closure (Or [Frag[A];Frag[B]]) matches [], [A], [B], [A;A], [A;B], [B;B], [B;A], and so on

- A ***matcher*** is a function that inspects a given fragment to find a match for a prefix that corresponds to a pattern
- It then checks whether the match is acceptable by testing whether a given acceptor succeeds on the corresponding suffix
- An ***acceptor*** is a function that accepts a fragment as an argument by returning some value wrapped inside the Some constructor.
- The acceptor rejects the fragment by returning None.

make_matcher

- make_matcher pattern returns a matcher for the pattern
- Matcher takes a fragment and an acceptor

Starting with the longest possible prefix:

Match the current prefix with the pattern
-> If match, check if the acceptor accepts
-> If acceptor accepts, return this prefix
-> If acceptor rejects, try to find another match for this prefix
-> If no match, remove the last element from the prefix and start over

make_matcher

```
let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher pat)
```

MATCHING *frag* FRAG [A; G; T]

```
| Frag frag -> make_appended_matchers match_nucleotide frag
```

```
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
```

```
let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None
```

```
let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)
```

MATCHING *LIST* LIST [FRAG [A; G]; JUNK 2]

```
| List pats -> make_appended_matchers make_matcher pats
```

- Same make_appended_matchers as previously, this time just used with make_matcher itself

```
let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher pat)
```

MATCHING Or. OR [FRAG [A; C]; FRAG [G; T]]

```
| Or pats -> make_or_matcher make_matcher pats
```

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

MATCHING *junk*

Junk 2

```
| Junk k -> match_junk k
```

```
let rec match_junk k frag accept =
  match accept frag with
    | None ->
        (if k = 0
         then None
         else match frag with
                | [] -> None
                | _::tail -> match_junk (k - 1) tail accept)
    | ok -> ok
```

MATCHING *closure* closure (FRAG [A; C])

```
| Closure pat -> match_star (make_matcher pat)
```

```
let rec match_star matcher frag accept =
  match accept frag with
    | None ->
        matcher frag
                (fun frag1 ->
                   if frag == frag1
                   then None
                   else match_star matcher frag1 accept)
    | ok -> ok
```

HOMEWORK 2

- Write a function make_matcher *gram* that returns a matcher for the grammar *gram*.
- When applied to an acceptor *accept* and a fragment *frag*, the matcher must return the first acceptable match of a prefix of *frag*, by trying the grammar rules in order

DEFINITIONS

- Fragment
  - A list of terminal symbols, e.g., ["3"; "+"; "4"; "-"].
- Derivation
  - A list of rules used to derive a phrase from a nonterminal.
- Prefix
  - [],[1],[1;2],[1;2;3] are prefix of [1;2;3]
- Suffix
  - [],[3],[2;3],[1;2;3] are prefix of [1;2;3]
- Matching Prefix
  - A prefix of a fragment that matches a derivation

DEFINITIONS

- Acceptor
  - A function whose argument is frag, if frag not accepted return None otherwise Some x.
- Matcher
  - A curried function with two args, acceptor and frag. Matcher matches prefix p of a frag such that accept accepts the corresponding suffix. If match, matcher returns what accept returns otherwise None.
- Parse Tree
  - A data structure which represents a parse tree is on the hw webpage. Similar to the binary tree type we talked about yesterday
- Parser
  - A function from fragments to parse trees

THINGS TO KEEP IN MIND

- Make use of recursion and pattern matching
- Make use of functions in List and Pervasives module
- Review slides from all discussions
- Run final code on SEASnet Linux servers. Make sure you are using the right version of Ocaml by checking path
- Ask questions on Piazza and come to Office hours
- Good luck! :)