CS131

Week 1

Saketh Kasibatla

Today

Recap Last Lecture

HW2

This Class

Homework 2 is due **one week from today at 23:55**

This homework will be graded with automated scripts

- not compiling → no credit
- function signatures must match, or no credit
- Look at the tests to figure out what function type signatures should be
- your code should behave exactly according to the spec
- if developing locally, test your homework out on SEASnet before submitting

Recap - Currying

Currying → calling a function with fewer arguments gives another function

```
let plus a b = a + b
val plus : int -> int -> int = <fun>

let plus5 = plus 5
val plus5 : int -> int = <fun>

let ans = plus5 10
val ans : int = 15

let plus = fun a -> fun b -> a + b
val plus : int -> int -> int = <fun>
```

Recap: Dictionaries (Solutions)

```
let d = [(hello, 5)]
val d : (string, int) list = [(hello, 5)]

type (‘key, ‘value) dict1 = (‘key, ‘value) list
type (‘key, ‘value) dict1 = (‘key, ‘value) list

let empty1 () = []
val empty1 : unit -> ‘a list

let dicta = empty1 ()
val dicta : ‘a list = []
```

Recap: Dictionaries (Solutions)

```
let put1 key value dict = (key, value)::dict
val put1 : ‘a -> ‘b -> (‘a, ‘b) list -> (‘a, ‘b) list = <fun>

let dictb = put1 42 “meow” dicta
val dictb : (int, string) list = [(42, “meow”)]

let rec get1 key dict = match dict with
  | [] -> None
  | (k, v)::xs -> if k = key
    then Some v
    else get1 key xs
val get1 : ‘a -> (‘a, ‘b) list -> ‘b option = <fun>

let ans = get1 42 dictb
val ans : string option = Some “meow”

let ans2 = get1 9 dictb
val ans2 : string option = None
```

Recap: Dictionaries (Solutions)

```
type (‘a, ‘b) dict2 = (‘a -> ‘b option)
type (‘a, ‘b) dict2 = (‘a -> ‘b option)

let empty2 () = fun requestKey -> None
val empty2 : unit -> ‘a -> ‘b option = <fun>

let empty_test1 = empty2 () 42 = None
val empty_test1 : bool = true

let emptyDict = empty2 ()
val emptyDict : (‘a, ‘b) dict2 = <fun>
```

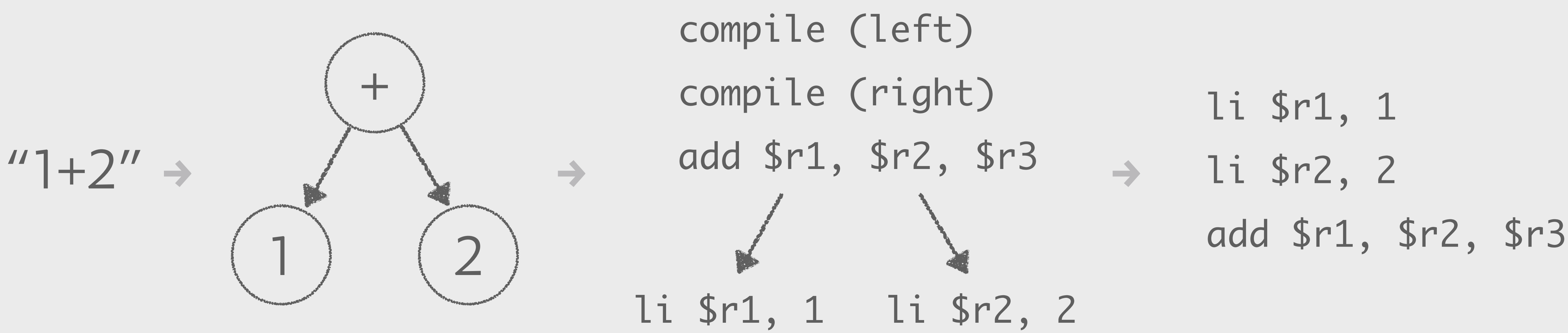
Recap: Dictionaries (Solutions)

```
let put2 key value dict = fun requestKey ->
  if requestKey = key
  then Some value
  else dict requestKey
val put2 : ‘a -> ‘b -> (‘a -> ‘b option) -> (‘a -> ‘b option) = <fun>

let a = put2 42 “meow” empty
val a : (int, string) dict2

let get2 requestKey dict = dict requestKey
val get2 : ‘a -> (‘a -> ‘b) -> ‘b

let b = get2 42 a
val b : string option = Some “meow”

let c = get2 9 a
val c : string option = None
```

“A programming system has two parts. The programming *environment* is the part that's installed on the computer. The programming *language* is the part that's installed in the programmer's head.”

–Bret Victor

How to Make a Language



How to Make a Language

plainText : string -(parser : string → AST)→ ast : AST

ast : AST -(interpreter : AST → output)→ output

OR

ast : AST -(compiler : AST → machineCode)→ machineCode

The *parser* enforces language *syntax*

syntax is in your head, *parser* is in the computer

HW2

make a parser generator

parser - takes a string and returns a derivation for that string

- string → derivation

parser generator - takes a grammar and gives you a parser

- grammar → parser
- grammar → string → derivation

sample solution

http://web.cs.ucla.edu/classes/fall17/cs131/hw/hw2-2006-4.html

Grammars (recap)

a way to describe which strings are and aren't valid for a language

symbol

- terminal - a symbol that can't be replaced with other symbols
- nonterminal - a symbol that can be replaced with other symbols

rule - a list of symbols that can replace a nonterminal symbol

grammar - a starting symbol and a set of rules that describe which symbols can be derived from a nonterminal

Grammars (HW1)

```
type ('nonterminal, 'terminal) symbol =
  | N of 'nonterminal
  | T of 'terminal
type ('nonterminal, 'terminal) symbol = N of 'nonterminal | T of ‘terminal

type rule = ‘nonterminal * (‘nonterminal, ‘terminal) symbol list
type rule = ‘nonterminal * (‘nonterminal, ‘terminal) symbol list

type grammar = ‘nonterminal * rule list
type grammar = ‘nonterminal *
  (‘nonterminal * (‘nonterminal, ‘terminal) symbol list) list
```

Grammars (HW2)

```
(Sentence,
  […;
    (Sentence, [N Quiet]);
    (Sentence, [N Grunt]);
    (Sentence, [N Shout]);
…])
```

```
(Sentence,
  function
  | …
  | Sentence -> [[N Quiet];
                 [N Grunt];
                 [N Shout]]
…)
```

Derivations

Phrase/Fragment: A list of terminal values e.g. `[“3”;“+”;“4”]`

```
type fragment = string list
```

Derivation: A list of rules that describe how to derive a phrase from a nonterminal symbol

```
type derivation = rule list
```

Derivations

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" .. "9"

Start: Expr

derivation for "3" "+" "4"

??

Derivations

Expr → Term Binop Expr
| Term

Term → Num
| Lvalue
| Incrop Lvalue
| Lvalue Incrop
| "(" Expr ")"

Lvalue → "$" Expr

Incrop → "++"
| "--"

Binop → "+"
| "-"

Num → "0" | "1" | "2" | "3"
| "4" | "5" | "6" | "7"
| "8" | "9"

Start: Expr

derivation for "3" "+" "4"

	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"

Leftmost derivation - fill in the left nonterminal first

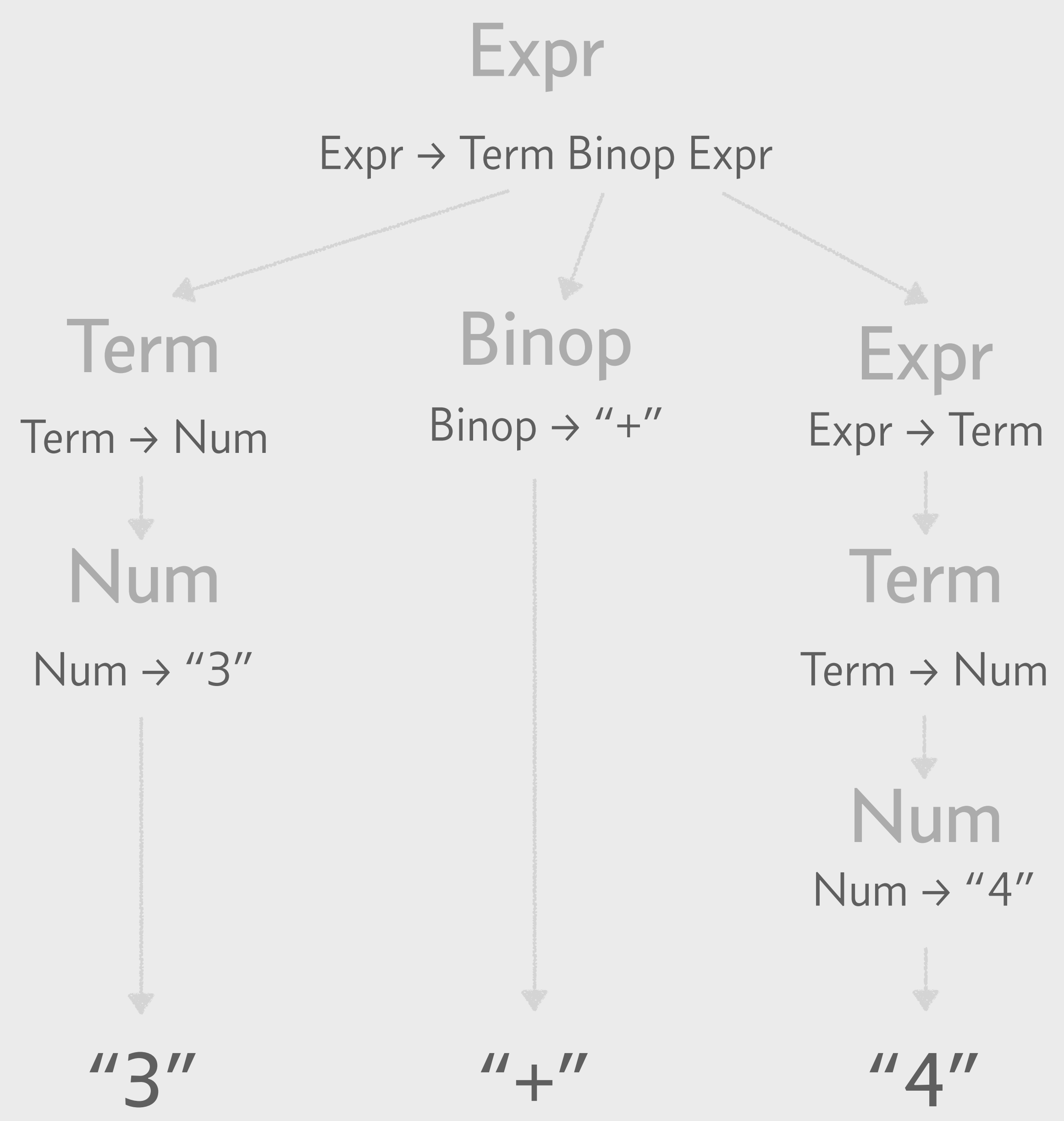
Derivations

```
let 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”]])
```

```
let fragment = [“3”; “+”; “4”]

let derivation = [
  Expr, [N Term; N Binop; N Expr];
  Term, [N Num];
  Num, [T “3”];
  Binop, [T “+”];
  Expr, [N Term];
  Term, [N Num];
  Num, [T “4”]
]
```

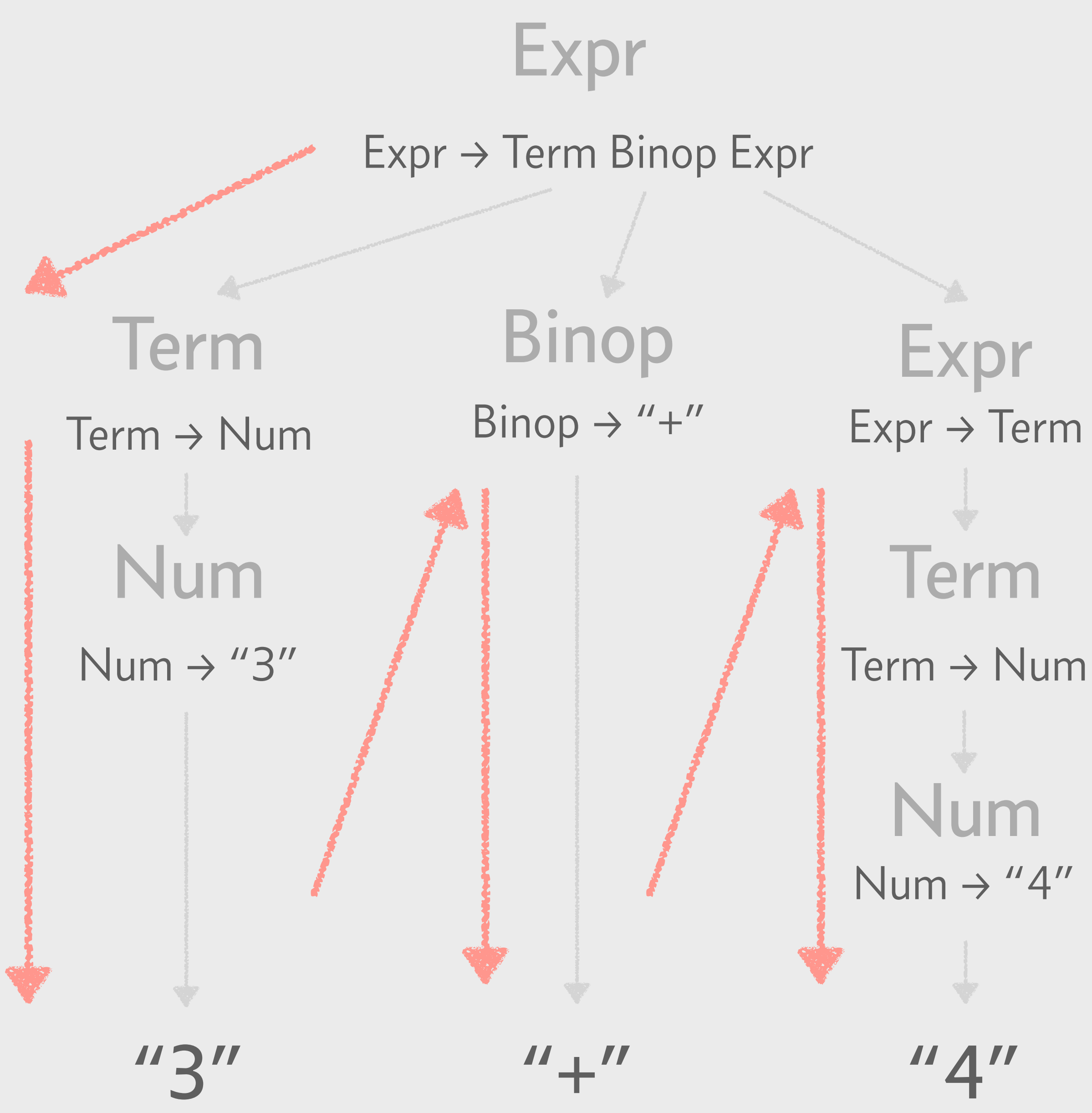
Leftmost Derivations

replace the leftmost nonterminal first



Leftmost Derivations

replace the leftmost nonterminal first



Matching Prefix

Prefix: [], [1], [1; 2], [1; 2; 3] are prefixes of [1; 2; 3]

Suffix: [1; 2; 3], [2; 3], [3], [] are the corresponding suffixes

Matching Prefix: a *prefix* of a fragment, for which there exists a derivation

Matching Prefix

Expr → Term Binop Expr
  | Term
Term → Num
  | Lvalue
  | Incrop Lvalue
  | Lvalue Incrop
  | "(" Expr ")"
Lvalue → "$" Expr
Incrop → "++"
  | "--"
Binop → "+"
  | "-"
Num → "0" .. "9"

Start: Expr

Find all matching prefixes of "3" "+" "2" "-" "6" in *grammar order*

grammar order - try rules as they are written in the grammar

Matching Prefix

```
Expr → Term Binop Expr
      | Term
Term → Num
      | Lvalue
      | Incrop Lvalue
      | Lvalue Incrop
      | "(" Expr ")"
Lvalue → "$" Expr
Incrop → "++"
       | "--"
Binop → "+"
      | "-"
Num → "0" .. "9"
```

Start: Expr

Find all matching prefixes of "3" "+" "2" "-" "6" in *grammar order*

1. "3" "+" "2" "-" "6"
2. "3" "+" "2"
3. "3"

Matcher

```
type matcher = fragment -> acceptor -> (derivation * suffix) option
```

1. Find next matching prefix in fragment

2. If no prefix found, return None

3. Else, call the acceptor with derivation, suffix

4. If acceptor returns None, go back to 1 else, return whatever acceptor returned

```
let match_empty frag accept = accept frag
match_empty [1; 2; 3] (fun s -> Some ([], s))
```

Acceptor

```
type acceptor = derivation -> suffix -> (derivation, suffix) option
```

If the acceptor rejects, then returns None

If the acceptor accepts, then returns Some (derivation, suffix)

- derivation, suffix could be same as inputs,
- but acceptor could also do further parsing

```
let accept_empty_suffix derivation suffix = function
| [] -> Some (derivation, [])
| _ -> None
val accept_empty_suffix : rule list -> suffix -> (rule list * suffix) option = <fun>

accept_empty_suffix [“RULE1”; “RULE2”] []
- : (string list * ‘a list) option = Some ([“RULE1”; “RULE2”], [])
```

Summary of Terms

Phrase/Fragment → A list of terminal symbols

Derivation → A list of rules that describe how to get a phrase from the start rule

Prefix → [], [1], [1; 2], [1; 2; 3] are prefixes of [1; 2; 3]

Suffix: → [1; 2; 3], [2; 3], [3], [] are the corresponding suffixes

Matching Prefix → A prefix of a fragment for which there exists a derivation

Acceptor → A function that takes a derivation and a suffix, and returns None or Some (derivation, suffix)

Matcher → A function that takes a fragment and an acceptor, and returns None or Some (derivation, suffix)

HW2

write a function

```
parse_prefix grammar acceptor fragment
```

which finds a derivation for a prefix of `fragment` (starting at `grammar`'s start rule) that `acceptor` will accept

```
(parse_prefix awkish_grammar accept_all [“9”, “+”, “2”]) =
Some ( [
  Expr, [N Term; N Binop; N Expr];
  Term, [N Num];
  Num, [T "9"];
  Binop, [T "+"];
  Expr, [N Term];
  Term, [N Num];
  Num, [T "2"]
], [])
```

HW2 Sample

```
(* http://web.cs.ucla.edu/classes/fall17/cs131/hw/hw2-2006-4.html *)
type nucleotide = A | C | G | T
type fragment = nucleotide list
type acceptor = fragment -> fragment option
type matcher = fragment -> acceptor -> fragment option

let accept_empty fragment = match fragment with
| [] -> Some []
| _ -> None

type pattern =
  | Frag of fragment
  | List of pattern list
  | Or of pattern list

let rec make_matcher = function
  | Frag frag -> make_fragment_matcher frag
  | List pats -> make_list_matcher pats
  | Or pats -> make_or_matcher pats
```

Fragment Matcher

```
let (rec) make_nucleotide_matcher nucleotide fragment accept = ?
val match_nucleotide : nucleotide -> nucleotide list ->
  (nucleotide list -> fragment option) -> fragment option = <fun>

let test1 = make_nucleotide_matcher A [A] accept_empty = Some []
val test1 : bool = true

let test2 = make_nucleotide_matcher G [A] accept_empty = None
val test2 : bool = true

let (rec) make_fragment_matcher frag fragment accept = ?
val match_fragment : nucleotide list -> nucleotide list ->
  (nucleotide list -> fragment option) -> fragment option = <fun>

let test3 = make_fragment_matcher [A; G; C] [A; G; C] accept_empty = Some []
val test3 : bool = true

let test4 = make_fragment_matcher [A; G] [A] accept_empty = None
val test4 : bool = true
```

List Matcher

```
let (rec) make_list_matcher list fragment accept = ?
val make_list_matcher : pattern list -> nucleotide list ->
  (nucleotide list -> fragment option) -> fragment option

let test1 = make_list_matcher [Frag [A; G]; Frag [C]] [A; G; C] accept_empty = Some []
val test1 : bool = true

let test2 = make_list_matcher [Frag [A; G]; Frag [C]] [A; G; C; A] accept_empty = None
val test2 : bool = true
```

Or Matcher

```
let (rec) make_or_matcher pats fragment accept = ?
val make_or_matcher : pattern list -> nucleotide list ->
  (nucleotide list -> fragment option) -> fragment option

let test1 = make_or_matcher [Frag [A; C]; Frag [G]] [G] accept_empty = Some []
val test1 : bool = true

let test2 = make_or_matcher [Frag [A; C]; Frag [G]] [A;C] accept_empty = Some []
val test2 : bool = true

let test3 = make_or_matcher [Frag [A; C]; Frag [G]] [T] accept_empty = None
val test3 : bool = true
```