CS131 - Week 2

UCLA Winter 2019
TA: Kimmo Karkkainen

Today

- OCaml recap and some new topics
- HW2

Slides are available on *Piazza* under *Resources* link

OCaml functions

```
let sum a b = a + b;;
```

```
# sum 1 2;;
- : int = 3
```

- **Remember**: no need for parentheses unless you want to specify the order of evaluation

```
# sum 1 2 * 3;;
- : int = 9
# sum 1 (2 * 3);;
- : int = 7
```

OCaml associativity and precedence

No need to memorize this, but good as a reference, can be found from documentation

Shows e.g. that function application happens before arithmetic operations (see example on previous slide)

Construction or operator	Associativity
prefix-symbol	–
`. .( .[ .{` (see section 8.15)	–
`#...`	–
function application, constructor application, tag application, `assert`, `lazy`	left
`- -.` (prefix)	–
`**... lsl lsr asr`	right
`*... /... %... mod land lor lxor`	left
`+... -...`	left
`::`	right
`@... ^...`	right
`=... <... >... |... &... $... !=`	left
`& &&`	right
`or ||`	right
`,`	–
`<- :=`	right
`if`	–
`;`	right
`let match fun function try`	–

Lambda functions (Anonymous functions)

```
(fun a -> a + 1) 5;;
- : int = 6
```

What is the relation between regular functions and lambda functions?

```
let add_one = (fun a -> a + 1);;
```

```
let add_one a = a + 1;;
```

These functions are exactly the same thing - the right one is converted into the left one. In both cases we are storing a function into a variable.

Functions with multiple arguments

- Professor told you that every function has only one argument - how is the following function possible?

```
let sum a b = a + b;;
```

Functions with multiple arguments

- Professor told you that every function has only one argument - how is the following function possible?

```
let sum a b = a + b;;
```

- This function gets converted to nested lambda functions:

```
let sum = (fun a -> (fun b -> a + b))
```

- This is also called ***currying*** - expressing a function with multiple arguments as multiple functions with single arguments

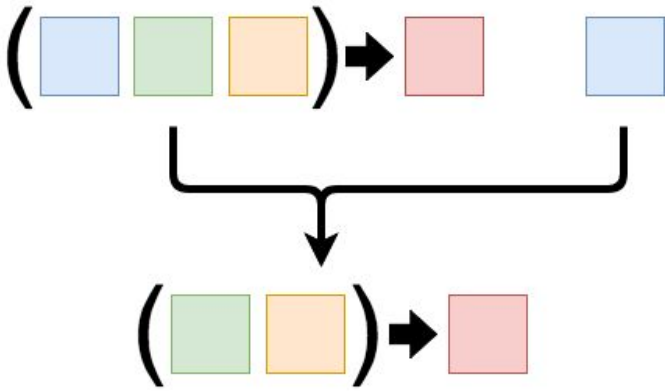
Currying & Partial application

- What's the benefit of nested functions?
- We can fix one of the arguments and return a new function:

```
let sum = (fun a -> (fun b -> a + b));;

let add_one = sum 1;;
```

- Now, *add_one* is equal to ***fun b -> 1 + b***



Currying - Function type

```
# let sum = (fun a -> (fun b -> a + b));;
val sum : int -> int -> int = <fun>

# let add_one = sum 1;;
val add_one : int -> int = <fun>
```

Understanding function types

- OCaml tries to figure out input and output types automatically:

```
# let sum a b = a + b;;
val sum : int -> int -> int = <fun>
```

Understanding function types

- OCaml tries to figure out input and output types automatically:

```
# let sum a b = a + b;;
val sum : int -> int -> int = <fun>
```

```
# let sum a b = a +. b;;
val sum : float -> float -> float = <fun>
```

Understanding function types

- Sometimes, the exact type is not known:

```
# let rec my_map func my_list = match my_list with
    | [] -> []
    | head::tail -> (func head) :: (my_map func tail);;

val my_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- In this case, ***'a*** and ***'b*** tell us that the argument is polymorphic
  - All instances of ***'a*** must match with each others, same applies to instances of ***'b***

Understanding function types

- After partial application, our types become fixed:

```
val my_map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# my_map (fun x -> x + 1);;
- : int list -> int list = <fun>
```

Recursive functions recap

- Need to add *rec* keyword:

```
let rec factorial = function
| x when x = 1 -> 1
| x -> x * factorial (x - 1);;
```

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

Mutual recursion

- Useful if you want to alternate recursing with two functions

```
let rec is_even = function
| x when x = 0 -> true
| x -> is_odd (x - 1)

and is_odd = function
| x when x = 0 -> false
| x -> is_even (x - 1);;
```

```
# is_even 10;;
- : bool = true
# is_odd 10;;
- : bool = false
# is_odd 9;;
- : bool = true
```

Infix functions

- Sometimes, defining your own infix operators might make your code simpler
- E.g. you'd like to concatenate strings by saying ***"abc"+"def"*** like in Python

```
let (+) a b = a ^ b;;

# "abc" + "def";;
- : string = "abcdef"
```

Infix functions

- Warning: Function definitions always override existing definitions - no *overloading*!

```
# 1 + 2;;
Error: This expression has type int but an expression was expected of type string
```

- This is why e.g. floating point operators are called differently (**+. -. /. *.**)
- Note that the infix operators are just regular functions with a special syntax

Infix functions

- You might also want to do the reverse, i.e. use an infix operator as a regular function:

```
# (+) 1 2;;
- : int = 3
```

- Useful when passing the function as a parameter or using partial application:

```
# List.map ((+) 1) [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]
```

=

```
# List.map (fun x -> x + 1) [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]
```

Function type exercises

What is the type of the following expression?

```
let rec f = function
  | 0 -> 1
  | x -> x * (f (x - 1));;
```

Function type exercises

What is the type of the following expression?

```
let f = (fun a -> fun b -> a*b);;
```

Function type exercises

What is the type of the following expression?

```
let f = function
| a,b when a > 5 -> b
| _ -> “No matches”□;;
```

Function type exercises

What is the type of the following expression?

```
let f f = f (f 1);;
```

Pattern matching recap

Three ways to do pattern matching:

```
# let first x = match x with
   | (left,_) -> left;;

# first (1,2);;
- : int = 1
```

```
# let first x = function
   | (left,_) -> left;;

# first (1,2);;
- : int = 1
```

```
# let first (left, _) = left;;

# first (1,2);;
- : int = 1
```

Pattern matching recap

- The last way allows matching with multiple arguments, but can't have multiple patterns for one argument or use *when* condition:

```
# let firsts (a,_) (b,_) = a,b;;

# firsts (1,2) (3,4);;
- : int * int = (1, 3)
```

Type Recap

- Built-in types:
  - int, float, char, string, bool, unit, tuple, list, function, ...
- Variant types:
  - Our type can be constructed using multiple different types
  - Useful e.g. when you need multiple types in one list

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

Optional type

- Sometimes your function can't come up with a return value
  - E.g. earlier example where we wanted to find contents for elements of type B, but compiler wanted us to handle type A also
- In these cases we could come up with some "fake" result (-1, empty list, etc)
- We could also throw an exception
- Another alternative is to use *Optional*, for example:

```
let divide a b = match a,b with
| x,y when y = 0.0 -> None
| x,y -> Some( x /. y );;
```

```
# divide 1.0 0.0;;
- : float option = None
# divide 1.0 2.0;;
- : float option = Some 0.5
```

Optional type

```
let divide a b = match a,b with
| x,y when y = 0.0 -> None
| x,y -> Some( x /. y );;
```

```
# divide 1.0 0.0;;
- : float option = None
# divide 1.0 2.0;;
- : float option = Some 0.5
```

- Downside: We have to handle this in the calling function:

```
let print_division a b = match (divide a b) with
| Some x -> print_float x
| None -> print_string "Undefined";;
```

```
# print_division 1.0 2.0;;
0.5
# print_division 1.0 0.0;;
Undefined
```

Recursive types

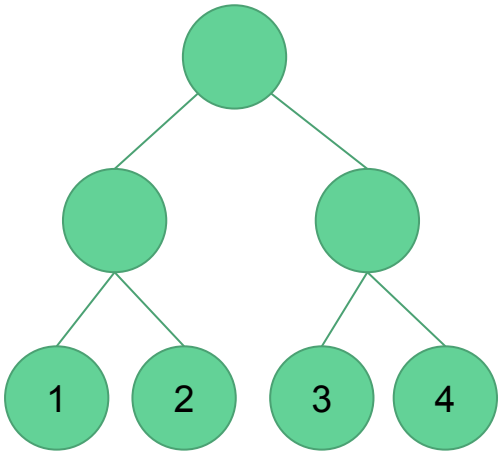
- We can use types to define recursive data structures, such as trees:

```
# type tree =
   | Leaf of int
   | Node of tree * tree;;

# let my_tree = Node (Node (Leaf 1, Leaf 2), Node (Leaf 3, Leaf 4))
```

```
# let rec first_leaf = function
| Leaf x -> x
| Node (left, right) -> first_leaf left;;

# first_leaf my_tree;;
- : int = 1
```



List module recap

- map
  - e.g. *List.map (fun x -> x*x) [1; 2; 3; 4; 5] -> [1; 4; 9; 16; 25]*
- filter
  - *List.filter (fun x -> x < 3) [1; 2; 3; 4; 5] -> [1; 2]*
- rev
  - *List.rev [1; 2; 3; 4; 5] -> [5; 4; 3; 2; 1]*
- for_all
  - *List.for_all (fun x -> x mod 2 = 0) [2; 4; 6] -> true*
  - *List.for_all (fun x -> x mod 2 = 0) [1; 4; 6] -> false*
- exists
  - *List.exists (fun x -> x mod 2 = 0) [1; 2; 3] -> true*

Note on List.filter

- Many students had trouble with the list type after using filter:

```
type ('a, 'b) my_type =
| A of 'a
| B of 'b;;

let my_list = [A 6; B "a"; B "c"];;

let is_B = function
| A _ -> false
| B _ -> true;;

let only_B = List.filter is_B my_list;;

List.iter print_string only_B;;
```

```
Error: This expression has type (int, string) my_type list
       but an expression was expected of type string list
       Type (int, string) my_type is not compatible with type string
```

Note on List.filter

- Many students had trouble with the list type after using filter:

```
type ('a, 'b) my_type =
| A of 'a
| B of 'b;;

let my_list = [A 6; B "a"; B "c"];;

let is_B = function
| A _ -> false
| B _ -> true;;

let only_B = List.filter is_B my_list;;

List.iter print_string only_B;;
```

```
Error: This expression has type (int, string) my_type list
       but an expression was expected of type string list
       Type (int, string) my_type is not compatible with type string
```

Filter does not change the list type!

So how can we fix this?

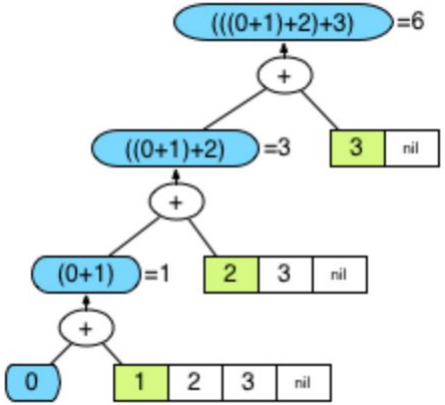
```
# let extract_B = function
  | B x -> x
  | _ -> raise (Failure "wrong type");;

# List.map extract_B only_B;;
- : string list = ["a"; "c"]
```

List module - fold_left

- Useful e.g. when you want to summarize a list into one value
- In some languages, this is also called *reduce*

```
# let sum my_list = List.fold_left (fun acc x -> acc + x) 0 my_list;;
# sum [1; 2; 3];;
- : int = 6
```



List module - fold_left

```
# let concat my_list = List.fold_left (fun acc str -> acc ^ str) "" my_list;;
# concat ["a"; "b"; "c"];;
- : string = "abc"
```

```
# let flatten my_list = List.fold_left (fun acc l -> acc @ l) [] my_list;;
# flatten [["a"; "b"]; ["c"; "d"]; ["e"; "f"]];;
- : string list = ["a"; "b"; "c"; "d"; "e"; "f"]
```

List module - Flatten

- The function we saw on the previous slide is also defined in the List module:

```
# List.flatten [["a"; "b"]; ["c"; "d"]; ["e"; "f"]];;
- : string list = ["a"; "b"; "c"; "d"; "e"; "f"]
```

List module - Mapi

- *mapi* is similar to *map*, except you can use the index of the element also

```
# List.mapi (fun i x -> (i, x)) ["a"; "b"; "c"; "d"; "e"];;

- : (int * string) list = [(0, "a"); (1, "b"); (2, "c"); (3, "d"); (4, "e")]
```

List module - Combine

- Combines elements from two lists to a list of tuples
- Many languages call this *zip*

```
# let first_names = ["Jon"; "Arya"; "Tyrion"];;
# let last_names = ["Snow"; "Stark"; "Lannister"];;
# let new_list = List.combine first_names last_names;;
val new_list : (string * string) list = [("Jon", "Snow"); ("Arya", "Stark"); ("Tyrion", "Lannister")]

# List.map (fun (first,second) -> first^" "^second) new_list;;
- : string list = ["Jon Snow"; "Arya Stark"; "Tyrion Lannister"]
```

List module - Split

- Opposite of *combine*

```
# List.split [("Jon", "Snow"); ("Arya", "Stark"); ("Tyrion", "Lannister")];;
- : string list * string list = (["Jon"; "Arya"; "Tyrion"], ["Snow"; "Stark"; "Lannister"])
```

List module exercises

How can we get list elements in positions 3-5?

List module exercises

How can we get the sum of hobbit ages:

```
let hobbits =
[("Bilbo", 129);
("Frodo", 51);
("Merry", 37);
("Pippin", 29)];;
```

→

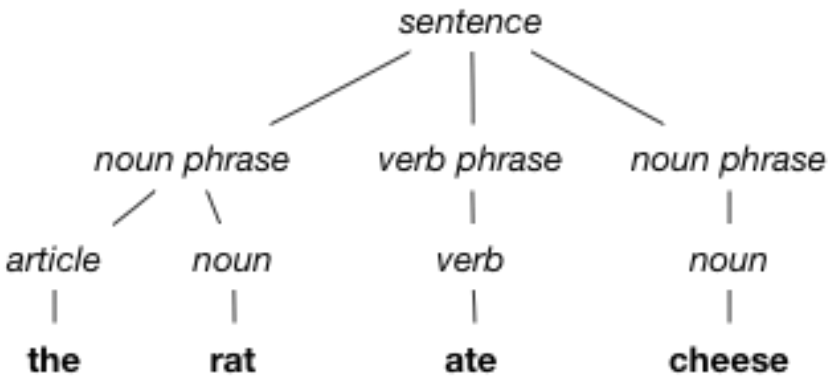
```
246
```

Homework 2

Homework 2

- Deadline Tuesday 1/29 (About 1.5 weeks from now)
- Significantly more difficult than the first homework!

- Main task: Find a parse tree for a given sentence



Grammars recap

- Grammar defines a language
  - What strings are valid sentences
- Consists of:
  - Rules
  - Non-terminal symbols
  - Terminal symbols
- Rules are applied by replacing non-terminal symbols using with the right-hand side of the rule

Example grammar:

PHRASE -> NOUN VERB
NOUN -> Mary
NOUN -> Mark
VERB -> eats
VERB -> drinks

Derivation

- Grammars are usually used to determine whether a string is valid and what the structure of it is
- Most common way is to use *top-down derivation*
  - Start from the *start symbol* and keep trying different rules in order
  - Try rules in order by replacing the left-most non-terminal symbol
- Example: Find a derivation for “Mark drinks”

Example grammar:

PHRASE -> NOUN VERB
NOUN -> Mary
NOUN -> Mark
VERB -> eats
VERB -> drinks

Derivation (Mark drinks)

Current sentence:
PHRASE
NOUN VERB
Mary VERB
NOUN VERB
Mark VERB
Mark eats
Mark VERB
Mark drinks

Rules applied:
PHRASE -> NOUN VERB
NOUN -> Mary
Backtrack
NOUN -> Mark
VERB -> eats
Backtrack
VERB -> drinks

Example grammar:

PHRASE -> NOUN VERB
NOUN -> Mary
NOUN -> Mark
VERB -> eats
VERB -> drinks

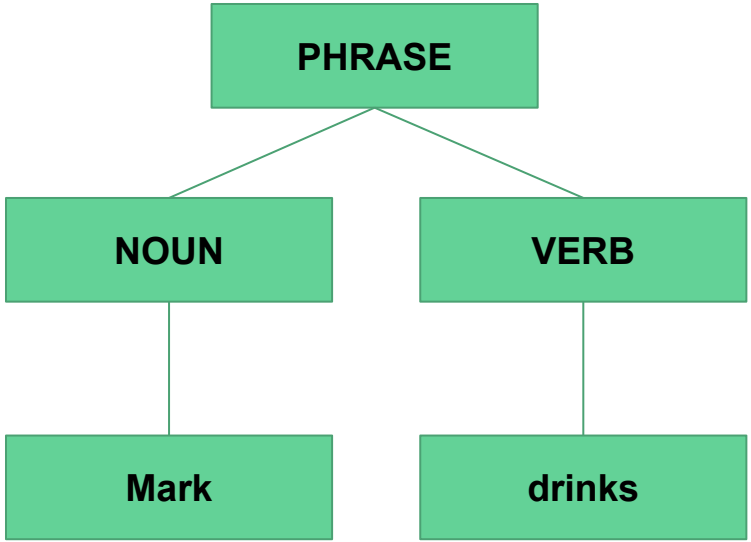
The derivation we found is:

PHRASE -> NOUN VERB
NOUN -> Mark
VERB -> drinks

Derivation tree / Parse tree

The derivation we found is:

PHRASE -> NOUN VERB
NOUN -> Mark
VERB -> drinks



Homework 2 - Problem 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"]])
```

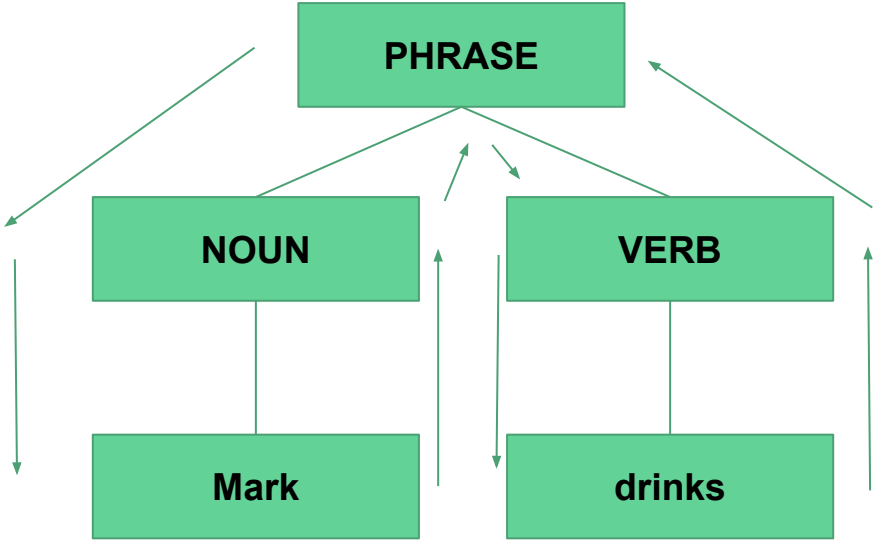
Homework 2 - Problem 1 (Warm-up)

- Note that your solution does not need to use pattern matching even though the example does! You need to write a function that takes a non-terminal symbol and returns a list of right-hand sides of rules

```
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 - Problem 2 (Warm-up)

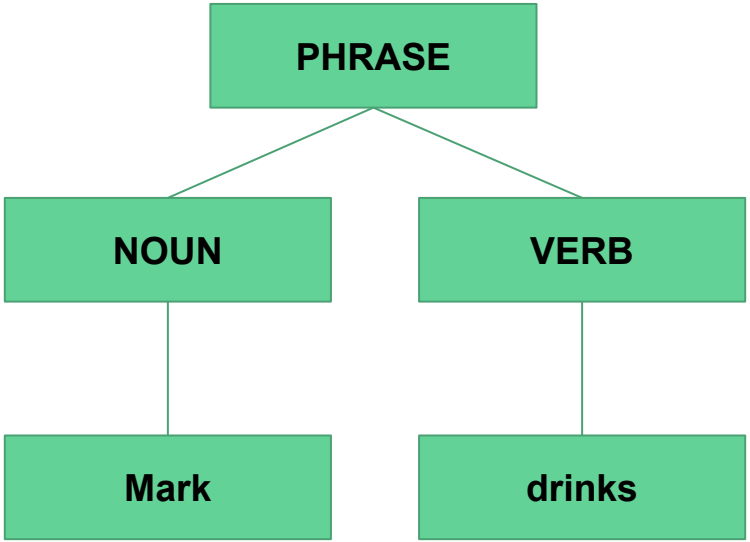
- Write a function *parse_tree_leaves tree* that returns a list of leaves in the tree from left to right (in the example below, return [“Mark”; “drinks”])



Homework 2 - Problem 2 (Warm-up)

- Tree is represented using data structure:

```
type ('nonterminal, 'terminal) parse_tree =
  | Node of 'nonterminal * ('nonterminal, 'terminal) parse_tree list
  | Leaf of 'terminal
```



Homework 2 - Problem 3

Write a function *make_matcher gram* that returns a matcher for the grammar *gram*

Terminology:

Fragment = List of terminals, e.g. [“Mark”; “eats”; “pizza”]

Suffix = List of terminals that were not used in the derivation, e.g. [“pizza”]

Acceptor = Function that takes a suffix and determines whether it is acceptable

E.g. *function | "pizza"::t -> Some ("pizza"::t) | _ -> None*

Matcher = Function that takes a fragment and an acceptor, and returns an optional parse tree (or *None* if there isn’t one that is accepted)

Homework 2 - Problem 3

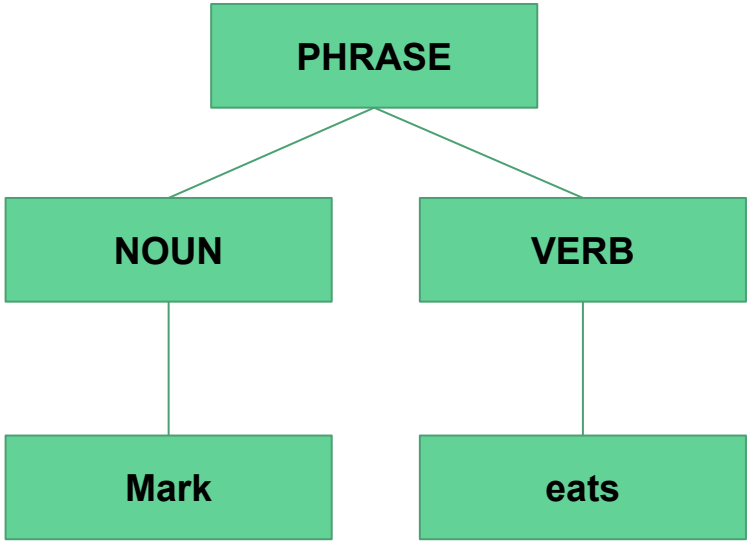
Outline for program:

1. Expand parse tree using the top-down derivation rules from the earlier slide
2. Once you have only terminal symbols and the symbols match with the input string, call the acceptor with the unmatched input symbols
   a. If the acceptor accepts, return that tree and you're done
   b. If the acceptor rejects, backtrack and try other rules

Homework 2 - Problem 3

Example:

1. We get input [“Mark”; “eats”; “pizza”]
2. Use grammar rules until there are no non-terminals and all the terminals match with our input (tree on the right)
3. Give unmatched suffix ([“pizza”]) to *acceptor*
4. Return parse tree if *acceptor* accepts it
    a. Otherwise, try other derivations



Homework 2 - Problem 4

Write a function *make_parser gram* that returns a parser for the grammar *gram*.

Parser is the same as matcher, except it takes only one input: **fragment**, and returns a parse tree only if there is no suffix left over.

E.g. Parsing *[“Mark”; “eats”; “pizza”]* must return *None*, as we can’t parse it without having *[“pizza”]* left as a suffix

Homework 2 - Problems 5-7

5. Write a non-trivial test case for *make_matcher*; this should inlude writing your own grammar

6. Using the grammar from problem 5, write a test case for *make_parser*

7. Write a report
    - Did you use *make_matcher* to write *make_parser* or vice versa or neither, and why?
    - Explain the weaknesses of your solution, provide test cases if possible
        - There will be some weaknesses, your parser/matcher might fail with a specific type of grammar for example

Homework 2 - Hint

An old homework is provided as a hint:
http://web.cs.ucla.edu/classes/winter19/cs131/hw/hw2-2006-4.html

This homework includes sample code for a similar problem, which you are allowed to use as a starting point

We'll discuss it in detail next week

Questions?