CS131 - Week 3

UCLA Winter 2019
TA: Kimmo Karkkainen

Today

- Currying & function types recap
- Grammars & derivations recap
- Old homework example
- HW2

Currying & Types recap

Currying

- What is currying?

Currying

- What is currying?
- Currying = Function with multiple arguments can be expressed as multiple functions with one argument each, e.g.:

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

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

Partial application

- What is partial application?

Partial application

- What is partial application?
- Partial application = Providing only part of the arguments, thereby fixing the values in the function and creating a function with fewer arguments, e.g.:

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

# let sum1 = sum 1;;
val sum1 : int -> int = <fun>

# sum1 2;;
- : int = 3
```

Partial application problems

What is the type of the following expression?

```
let rec f a = function
  | x when x mod a = 0 -> true
  | _ -> false;;
```

Partial application problems

What is the type of the following expression?

```
let rec f a = function
  | x when x mod a = 0 -> true
  | _ -> false;;
```

And after partial application? Also, what does this function do?

```
let g = f 2;;
```

Partial application problems

What is the type of the following expression?

```
let f a b = a b;;
```

Partial application problems

What is the type of the following expression?

```
let f a b = a b;;
```

How about after partial application? Also, what does this function do?

```
# f (fun x -> x + 1);;
```

Grammar recap

Derivation recap

- 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 recap

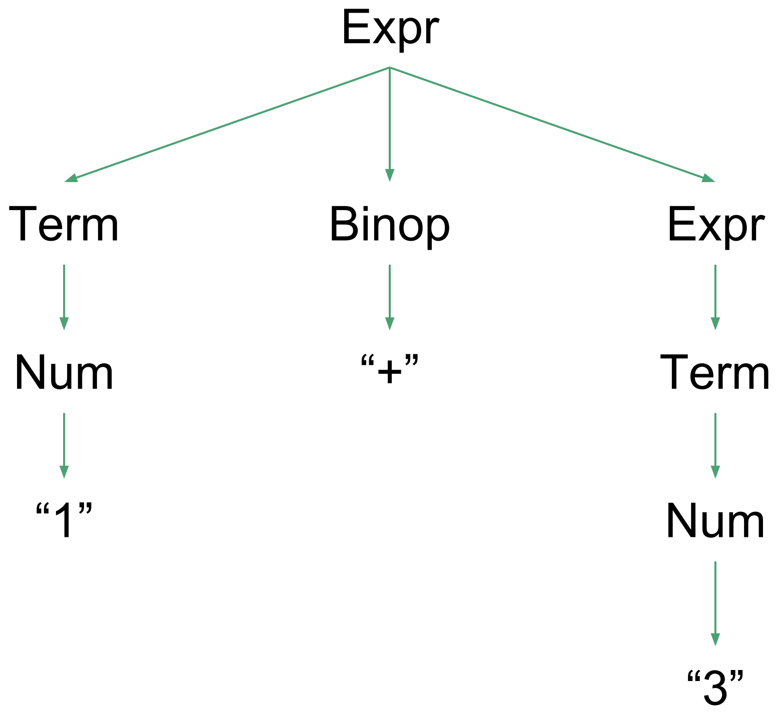
- Recap of the top-down parsing technique:
  How to derive 1+3?

Current symbols:	**Rule applied:**
Expr	Expr -> Term Binop Expr
Term Binop Expr	Term -> Num
Num Binop Expr	Num -> “0”
“0” Binop Expr	Backtrack
Num Binop Expr	Num -> “1”
“1” Binop Expr	Binop -> “+”
“1” “+” Expr	Expr -> Term Binop Expr
“1” “+” Term Binop Expr	… Fails eventually with Binop, backtrack …
“1” “+” Expr	Expr -> Term
“1” “+” Term	Term -> Num
“1” “+” Num	… Try all Num rules until “3” …
“1” “+” “3”	Done!

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 recap - Parse tree for [“1”; “+”; “3”]

```
Node (Expr, [
        Node (Term, [
                Node (Num, [
                        (Leaf "1")
                ])
        ]);
        Node (Binop, [
                Leaf "+"
        ]);
        Node (Expr, [
                Node (Term, [
                        Node (Num,
                                [Leaf "3"])
                ])
        ])
])
```



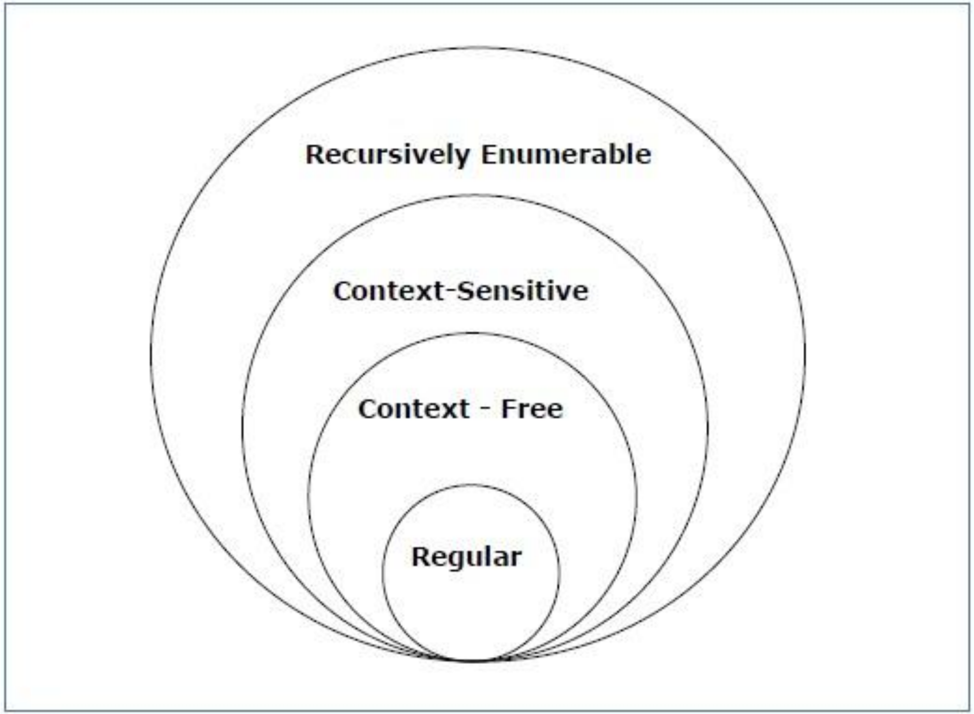
Old homework example

Old homework example

- You can use code from an old homework as the starting point for your solution: http://web.cs.ucla.edu/classes/winter19/cs131/hw/hw2-2006-4.html
- Goal: Build a pattern matcher for genetic sequences (similar to regular expressions)
- 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)

Context-Free vs Regular Grammar

- Your homework deals with context-free grammars, the old homework example uses a regular grammar
- Regular grammars only allow rules of type S -> aS | a | ε
- Context-free grammars allow e.g. palindromes: S -> aSa | b
- Discussed more in CS181



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;G], [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

Pattern matching

- How to match *AAGC* using the following pattern?
- List [

```
    Frag [A];
    Or [
        Frag[T];
        Junk 2;
    ];
    Frag [G; C]]
```

Pattern to context-free grammar - example

- Frag [A; T; G; C]

Pattern to context-free grammar - example

- Or [Frag [A; T]; Frag [G; C]; Frag [G; T]]

Pattern to context-free grammar - example

- Closure (Frag [“A”; “T”])

Pattern to context-free grammar - example

- List

```
[
  Frag [A; T];
  Junk 1;
]
```

make_matcher

- *make_matcher pattern* returns a matcher for *pattern*
- Matcher takes a fragment and an acceptor, returns whatever the acceptor returns
- Similar to your matcher, except instead of a context-free grammar, it matches more limited patterns

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

Matching *Frag*

Frag [A; G; T]

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

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

```
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
```

Matching *Frag*

Frag [A; G; T]

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

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

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

Matching *Frag*

- Make_appended_matchers creates a chain of matchers
- Assuming our matcher tries to match Frag [A; G; T] with input [A; G; T]:

match_nucleotide (A)
Compare the first input symbol A to A

“acceptor”
Call next match_nucleotide with remaining input symbols [G; T]

match_nucleotide (G)
Compare the first input symbol G to G

“acceptor”
Call next match_nucleotide with remaining input symbols

match_nucleotide (T)
Compare the first input symbol T to T

“acceptor”
Call next match_nucleotide with remaining input symbols

match_nucleotide
Notice pattern is empty -> match_empty

match_empty
Call the original acceptor with remaining input symbols

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
```

Old homework example

- Note the differences to your homework:
  - You need to match arbitrary context-free grammars, so the solution should be more abstract
    - No need for specialized functions for all the different cases
  - In your parser, you need to return the parse tree instead of what the acceptor returns
- Consider especially how make_appended_matchers and make_or_matcher are related to your homework
  - Other functions are related to specific patterns and are probably less useful to you

HW2 Recap

Homework #2 - Part 1

- Transforming grammars used in HW #1 to a new format
    - New format is easier to use inside the parser

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"]])
```

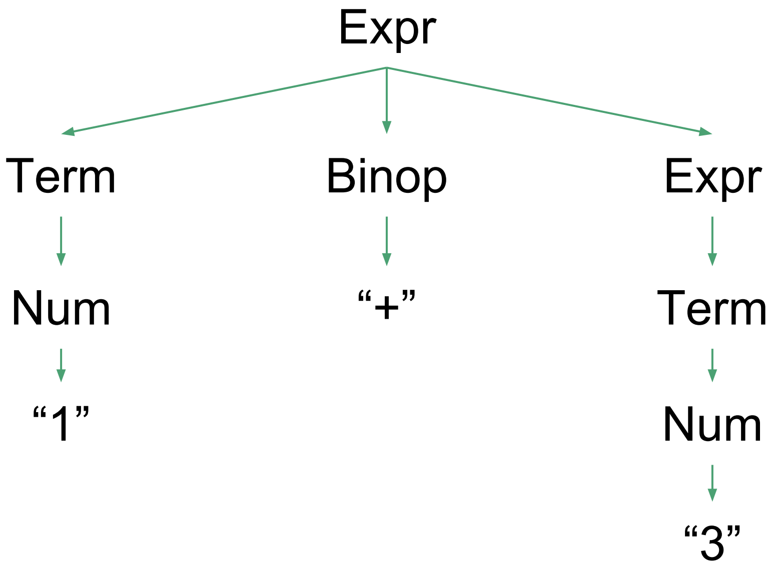
Homework #2 - Part 1

- Define a function that takes a non-terminal value and goes through the original list of rules, returning only the matching rules
- Hint: Combining functions from the List module makes this problem quite easy
- Hint: You don't need to use pattern matching in your solution, as long as the function output is the same

```
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 [“1”, “+”, “3”])



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. [“1”; “+”; “2”; “-”]

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

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 **whatever the acceptor returns** (or *None* if there isn’t one that is accepted)

Homework 2 - Problem 3 - Acceptors

Note that you are not expected to write the acceptors - they will be given as an argument. For example, in the given test cases your matcher will be used with:

```
let accept_all string = Some string

let accept_empty_suffix = function
   | _::_ -> None
   | x -> Some x
```

Homework 2 - Problem 3

Outline for program:

1. Expand non-terminals using the top-down derivation rules from the earlier slide
2. Once you have only terminal symbols and the symbols match with any prefix of the input string, call the acceptor with the unmatched input symbols (the suffix)
   a. If the acceptor accepts, return whatever the acceptor returns and you're done
   b. If the acceptor rejects, backtrack and try other rules

Homework 2 - Problem 3

Example:

Try to match an input [“a”; “b”] with a trivial grammar, assume we are using accept_empty_suffix acceptor:

```
S -> “a”
S -> “a” “b”
```

```
let accept_empty_suffix = function
   | _::_ -> None
   | x -> Some x
```

Homework 2 - Problem 3

Example:

Try to match an input [“a”; “b”] with a trivial grammar, assume we are using accept_empty_suffix acceptor:

```
S -> “a”
S -> “a” “b”
```

```
let accept_empty_suffix = function
   | _::_ -> None
   | x -> Some x
```

1. Try rule S -> “a”. This matches with a prefix [“a”] in our input, leaving a suffix [“b”].
2. Call the acceptor with the suffix [“b”]
   a. If acceptor return Some -> return that value and you’re done
   b. If acceptor returns None -> backtrack and try the next rule (S -> “a” “b”)

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 for the following differences:

1. Parser takes only the **fragment** as an input, no acceptor!
2. Parser will return a value only if it finds a derivation for the **entire input**
3. Parser returns a **parse tree** instead of the suffix (there's never a non-empty 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 - Notes

- Efficiency is not the primary criteria for grading, but note that we can't spend an hour testing your code
    - Very basic optimizations should be enough, e.g. do not search for all possible matches before calling the acceptor
- Before implementing anything, consider how you can build the parse tree
    - Adding it later might require major changes...
- Read & understand how *make_appended_matchers* and *make_or_matchers* work in the earlier solution - they might be useful in your own solution
    - Try checking the types of different functions - there's a lot of currying happening

Questions?