CS 131
PROGRAMMING LANGUAGES
(WEEK 3)

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
   Q1 & Q2
2. MATCHERS AND ACCEPTORS
3. CREATING OUR OWN MATCHERS
4. DISCUSSING HOMEWORK 2

HW 2 - TASK 1 - WARM UP

- Discussed last friday
- Format of grammar is different in HW 2 compared to HW 1
- Write a function `convert_grammar gram1` that takes HW1-style grammar and returns HW2-style grammar
- In HW1 we had List of Tuples for rules
- Converted grammar rules should return a function that has the corresponding matchings.

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"]])
```

CONVERT_GRAM TYPE

```
val awkish_grammar :
  awksub_nonterminals *
  (awksub_nonterminals -> (awksub_nonterminals, string) symbol list list) =
  (Expr, <fun>)
```

PARSE TREES

- Q2: Write a function parse_tree_leaves *tree* that traverses the parse tree *tree* left to right and yields a list of the leaves encountered.
- Def: A **parse tree** or **parsing tree** or derivation **tree** is an ordered, rooted **tree** that represents the syntactic structure of a string according to some context-free grammar.
- Function type:

```
val parse_tree_leaves : ('a, 'b) parse_tree -> 'b list = <fun>
```

SOME DEFINITIONS

- **Fragment**
  - A list of terminal symbols, e.g., ["3"; "+"; "4"; "-"].
- **Derivation**
  - A list of rules used to derive a phrase from a nonterminal.
- **Leftmost Derivation**
  - The leftmost nonterminal is always expanded next

DERIVATION EXAMPLE

Example: Derivation of 3+4:

```
[Expr, [N Term; N Binop; N Expr];
 Term, [N Num];
 Num, [T "3"];
 Binop, [T "+"];
 Expr, [N Term];
 Term, [N Num];
 Num, [T "4"]]
```

rule	after rule is applied
(at 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"

MORE DEFINITIONS

- Prefix:
  - [], [1], [1;2], [1;2;3] are prefixes of [1;2;3] (not necessarily in that order)
- Suffix:
  - [], [3], [2;3], [1;2;3] are suffixes of [1;2;3] (not necessarily in that order)
- Matching Prefix
  - A prefix of a fragment, for which there exists a derivation

MATCHING PREFIX (EXAMPLE)

- Find all matching prefixes of “3 + 2 - 6” in order:
- Answer:
  - First matching prefix: 3 + 2 - 6
  - Second matching prefix: 3 + 2
  - Third matching prefix: 3

<u>Grammar (Exp = Start Symbol)</u>

E -> E Op T | T

Op -> “+” | “-”

T -> “0” | “1” | … | “9”

MATCHERS

- Tries to create a derivation tree where the leaves are the string we are trying to match.
- We match the prefix with our grammar (in order)
- Whatever is not matched is called suffix.

ACCEPTOR

- Acceptor
  - A **function** that takes a fragment as argument and returns
    - None if rejected
    - Else Some x i.e. a value wrapped in an option
- You don’t have to write these as part of any Tasks, given in sample test code.
- You can create any acceptor you want.

ACCEPTOR (EXAMPLES)

```
let accept all string = Some string
```

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

```
utop # let accept_all string = Some string;;
val accept_all : 'a -> 'a option = <fun>
```

```
utop # let accept_empty_suffix = function
    | _::_ -> None
    | x -> Some x ;;
val accept_empty_suffix : 'a list -> 'a list option = <fun>
```

make_matcher

- A curried function that takes a fragment and an acceptor as arguments.

- Steps:
  a. Finds the next matching prefix
     - If no prefix matches -> return None
  b. Call Acceptor on suffix
     - If acceptor returns some value, return that
     - Else back to step a./

HW2 - EXAMPLE

- **awkish_grammar** [“3”; “+”; “4”; “-”]
- Matcher matches [“3”; “+”; “4”;]
- Acceptor takes the remaining suffix [“-”]
- Matcher matches [“3”]
  - Acceptor takes [“+”; “4”; “-”]

LETS MAKE SOME MATCHERS!

OR matcher

```
utop # let rec match_or l accept frag=
match l with
|f::r -> let mf = f accept frag
in if mf = None
then match_or r accept frag
else mf
|[] -> None;;
val match_or : ('a -> 'b -> 'c option) list -> 'a -> 'b -> 'c option = <fun>
```

SINGLE ELEMENT MATCHER

```
utop # let msm n accept frag=
match frag with
| h::t -> if h=n
then accept t
else None
|[] -> None;;
val msm : 'a -> ('a list -> 'b option) -> 'a list -> 'b option = <fun>
```

- Match “3” in ["3";"4";"5”]

```
utop # msm "3" accept_all ["3";"4";"5"];;
- : string list option = Some ["4"; "5"]
```

- Match “3 4” in ["3";"4";"5”]

```
utop # msm "3" (msm "4" accept_all) ["3";"4";"5"];;
- : string list option = Some ["5"]
```

HW2 :

```
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"]])
```

TEST CASES

```
let test0 =
  ((make_matcher awkish_grammar accept_all ["ouch"]) = None)

let test1 =
  ((make_matcher awkish_grammar accept_all ["9"])
   = Some [])

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

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

(* This one might take a bit longer.... *)
let test4 =
 ((make_matcher awkish_grammar accept_all
     ["("; "$"; "8"; ")"; "-"; "$"; "++"; "$"; "--"; "$"; "9"; "+";
      "("; "$"; "++"; "$"; "2"; "+"; "("; "8"; ")"; "-"; "9"; ")";
      "-"; "("; "$"; "$"; "$"; "$"; "$"; "++"; "$"; "$"; "5"; "++";
      "++"; "--"; ")"; "-"; "++"; "$"; "$"; "("; "$"; "8"; "++"; ")";
      "++"; "+"; "0"])
  = Some [])
```

Worked out on blackboard.

make_parser

- Takes grammar and makes a parse tree.
- Creates parse tree for the entire fragment.
- If *frag* cannot be parsed entirely (that is, from beginning to end), the parser returns None.
- Otherwise, it returns Some *tree* where *tree* is the parse tree corresponding to the input fragment.

TEST CASE

```
let test6 =
  ((make_parser awkish_grammar small_awk_frag)
   = Some (Node (Expr,
                [Node (Term,
                       [Node (Lvalue,
                              [Leaf "$";
                               Node (Expr,
                                     [Node (Term,
                                            [Node (Num,
                                                   [Leaf "1"])])])]);
                        Node (Incrop, [Leaf "++"])]);
                 Node (Binop,
                       [Leaf "-"]);
                 Node (Term,
                       [Node (Num,
                              [Leaf "2"])])])))
```

Worked out on blackboard.

TEST CASE

```
let test7 =
  match make_parser awkish_grammar small_awk_frag with
    | Some tree -> parse_tree_leaves tree = small_awk_frag
    | _ -> false
```

- The leaves of the parse tree (output of parse_tree_leaves) should be equivalent to the fragment you are parsing.

- Write a report explaining the design choices you made.
- Discuss grammars that may not work with your solution.
  - You are not expected to solve every single grammar, but you should write about what won’t work and why it won’t work.

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! :)