CS 131 - Week 2

TA : Tanmay Sardesai

How to find these slides

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

Announcements

- Email tanmays@cs.ucla.edu
- Office Hours Thursday 1:30 pm - 3:30 pm. Bolter Hall 3256S-B
- HW2 Due Tuesday 01/29 11:55 pm
    - Posted to http://web.cs.ucla.edu/classes/winter19/cs131/homework.html by EOD
    - Make sure to use the same function signatures
    - Follow all instructions
    - Submit ALL the files
    - Make sure code compiles
    - Submit on ccle

Topics covered today

- Review of last time
- Currying
- Grammar
- HW2
- Questions and Hands on Ocaml

Last Class

- Variables
- Lists
- Functions
- Recursive Functions
- Anonymous Functions
- Higher order functions
- Types in Ocaml
- Pattern matching
- Grammar

Currying

- Break a function with multiple arguments into functions that take a single argument.

- let sum a b = a + b;;
- let sum = fun a b -> a + b;;
- let sum = fun a -> fun b -> a + b;;

Currying

- Break a function with multiple arguments into functions that take a single argument.

- let sum a b = a + b;;
- let sum = fun a b -> a + b;;
- let sum = fun a -> fun b -> a + b;;

What is sum?

What is `sum 5 2`?

What is `sum 5`?

Context Free Grammar and HW 2 Discussion

- 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

Example of Grammar

Symbols: E, T, F, *, N,0,1,2,3,4,5,6,7,8,9, (,)

Non-Terminals: E, T, F, N

Terminals: *,0,1,2,3,4,5,6,7,8,9,(,)

Starting Symbol: E

Rules:

```
    E -> E + T
    E -> T
    T -> T*F
    T -> F
    F -> (E)
    F -> N
    N -> 0
    N -> 1
    ….
    N -> 9
```

Rules Abbr:

```
    E -> E+T | T
    T -> T*F | F
    F -> (E) | N
    N -> 0 | 1 | 2 | … | 9
```

Example Derivation

3 + 4

Rule	After it is applied
start	E
E	E + T
E + T	T + T
T + T	F + T
F + T	N + T
N + T	3 + T
3 + T	3 + F
3 + F	3 + N
3 + N	3 + 4

3 + (4 * 8)

start	E
E	E + T
E + T	T + T
T + T	F + T
F + T	N + T
N + T	3 + T
3 + T	3 + F
3 + F	3 + (E)
3 + (E)	3 + (T)
3 + (T)	3 + (T * F)
3 + (T * F)	3 + (F * F)
3 + (F * F)	3 + (N * F)
3 + (N * F)	3 + (4 * F)
3 + (4 * F)	3 + (4* N)
3 + (4 * N)	3 + (4 * 8)

Blind Alley Rules

- Any rule from which it is impossible to derive a string of terminals
- Example

Symbols: S, A, B, a, b
Non-Terminals: S, A, B
Terminals: a, b
Starting symbol: S,

Rules:
S -> A | B
A -> A | aB | aA | a
B -> B

What are the blind alley rules?

HW2 - 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.
- 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
- Acceptor
  - A function whose value is frag, if frag not accepted return None otherwise Some x.

HW2 - Some 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

HW 2 - Task 1

- 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

HW 1:

```
(Expr, [Expr, [N Term; N Binop; N Expr];
    Expr, [N Term];
    Term, [N Num];
    Term, [N Lvalue];
    Term, [N Incrop; N Lvalue];
    Term, [N Lvalue; N Incrop];
    Term, [T"("; N Expr; T")"];
    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"]])
```

HW2:

```
(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"]])
```

HW 2 - Task 1 - convert_gram

- In HW1 we had List of Tuples for rules
- Converted grammar rules should return a function such that when you pass some non-terminal as an argument it returns a list of lists which is all the rules for that argument

HW 2 - Task 2 - parse_tree_leaves

- Given a parse tree traverse it left to right and output a list of leaves

HW 2 - Task 3 - make_matcher

- Recall definitions
    - Acceptor
        - A function that takes fragment and returns
            - None if rejected
            - Else Some x
    - You don’t have to write these as part of any tasks. Required for testing
    - What do these do? Why do we need Some?

```
let accept_all string = Some string

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

HW 2 - Task 3

- Matcher
  a. A function that takes fragment and acceptor. Returns first acceptable matching prefix
- Steps:
  a. Find a matching prefix
     - If no matching, return None
  b. Call Acceptor with suffix
     - If acceptor returns some value, return that
     - Else back to step a

HW 2 - Task 3 - make_matcher

- Finally…
- Task 3 is to write a function which takes a grammar as an argument and returns a matcher.
- Function signature - make_matcher grammar that returns a matcher
- When the matcher is applied to a acceptor and fragment it returns - (the first matching prefix or acceptor suffix; specs unclear will be updated soon)
- Example:
    - [“3”,”+”,”4”,”-”] - **matcher** takes [“3”,”+”,”4”] and acceptor takes [“-”]. If we are using accept_all we return Some if we are using accept_empty we return None

HW 2 - Task 4 - make_parser gram

- Write a function that takes a grammar and returns a parser for that grammar. When applied to a fragment the parser returns a optional parse tree
- If a fragment cannot be passed the parser returns None
- Otherwise return Some tree where the tree is parse_tree for the fragment
- Should use rules in the same order as make_matcher

HW 2 - Task 5

- 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! :)

Hands On With Questions on everything functional

https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-core.html

https://ocaml.org/learn/tutorials/functional_programming.html