Midterm 2 Study Guide Solutions

Handout 27
CSCI 334: Spring 2024

Solutions

Q1. (10 points) .... Terminology

You should do this exercise on your own.

Q2. (10 points) .... Decision Problem

We prove that `bothHalt` is not computable by contradiction. Assume `bothHalt` exists. Then we can write the following function.

```
let halt p i =
  bothHalt p p i
```

If `p` halts, then `bothHalt` will return `true` because "both" `p` programs halt. Likewise, if `p` does not halt, then `bothHalt` will return `false` because "both" `p` programs do not halt. Since we are able to construct a `halt` function using the ordinary rules of logic (F#) with the only assumption being the existence of `bothHalt`, and we know that `halt` is not computable, then `bothHalt` must not be computable.

An interesting alternative raised in class is to define a different function, like so

```
let halt p i =
  let f x = x
  bothHalt p f i
```

This also works because when `p` halts, `f` also halts (because it always halts), so `bothHalt` returns `true`. If `p` does not halt, `bothHalts` returns `false` because it is sufficient for either program to not halt for `bothHalt` to return `false`. Good thinking!

Q3. (10 points) .... Parsing and Evaluation

Here is one possible solution.

```
open Combinator

type Expr =
| Increment
| Decrement
| MoveLeft
| MoveRight
| Print

type State = { tape: int list; pos: int }

let op =
    (
        (pchar '+' |>> fun _ -> Increment) <|>
        (pchar '-' |>> fun _ -> Decrement) <|>
        (pchar '<' |>> fun _ -> MoveLeft) <|>
```

```
        (pchar '>' |>> fun _ -> MoveRight) <|>
        (pchar 'p' |>> fun _ -> Print)
    ) <!> "op"

let expr = pmany0 op <!> "expr"

let grammar = pleft expr peof <!> "grammar"

let rec getAtIndex xs i =
    match xs with
    | [] -> failwith "Cannot find index in list."
    | y::_ when i = 0 -> y
    | _::ys -> getAtIndex ys (i - 1)

let rec updateAtIndex xs i v =
    match xs with
    | [] when i <> -1 -> failwith "Cannot find index in list."
    | [] -> []
    | _::ys when i = 0 -> v::ys
    | y::ys -> y::updateAtIndex ys (i - 1) v

let parse input =
    let i = prepare input
    match grammar i with
    | Success(ast,_) -> Some ast
    | Failure(_,_) -> None

let ephsPrint v =
    let s =
        match v with
        | 0 -> "a"
        | 1 -> "b"
        | 2 -> "c"
        | 3 -> "d"
        | 4 -> "e"
        | 5 -> "f"
        | 6 -> "g"
        | 7 -> "h"
        | 8 -> "i"
        | 9 -> "j"
        | 10 -> "k"
        | 11 -> "l"
        | 12 -> "m"
        | 13 -> "n"
        | 14 -> "o"
        | 15 -> "p"
        | 16 -> "q"
        | 17 -> "r"
        | 18 -> "s"
        | 19 -> "t"
        | 20 -> "u"
        | 21 -> "v"
        | 22 -> "w"
        | 23 -> "x"
```

```
        | 24 -> "y"
        | 25 -> "z"
        | _ -> "poo"
    printf "%s" s

let evalOp (e: Expr)(s: State) : State =
    match e with
    | Increment ->
        let cur = getAtIndex s.tape s.pos
        let upd = cur + 1
        let tape' = updateAtIndex s.tape s.pos upd
        { s with tape = tape' }
    | Decrement ->
        let cur = getAtIndex s.tape s.pos
        let upd = cur - 1
        let tape' = updateAtIndex s.tape s.pos upd
        { s with tape = tape' }
    | MoveLeft ->
        if s.pos > 0 then
            { tape = s.tape; pos = s.pos - 1 }
        else
            printfn "Tape head out-of-bounds (on left)."
            exit 1
    | MoveRight ->
        if s.pos < 10 then
            { tape = s.tape; pos = s.pos + 1 }
        else
            printfn "Tape head out-of-bounds (on right)."
            exit 1
    | Print ->
        let cur = getAtIndex s.tape s.pos
        ephsPrint cur
        s

let rec eval (es: Expr list)(s: State) : State =
    match es with
    | [] -> s
    | op::ops ->
        let s' = evalOp op s
        eval ops s'

[<EntryPoint>]
let main args =
    let file = args[0]
    let input = System.IO.File.ReadAllText file
    let ast_maybe = parse input
    match ast_maybe with
    | Some ast ->
        eval ast { tape = [0;0;0;0;0;0;0;0;0;0]; pos = 0 } |> ignore
        0
    | None ->
        printfn "Invalid program"
        1
```

Q4. (10 points) .... Partial and Total Functions

(a) Partial function: $\{\langle n, \texttt{fibonacci}\ n\rangle \mid n \in \mathbb{Z} \wedge x \geq 0\}$

(b) Looking at our `gcd` function, the only operation that may be problematic is `%`. The answer I gave in class assumed $\mod$ was undefined for negative numbers, but actually, I had it backward: $\mod$ is defined for all divisors and quotients in $\mathbb{Z}$. Some programming languages, like C, assume that the quotient and divisor are not negative, and will give the wrong answer otherwise. If we stick with the mathy definition (instead of C), then `gcd` appears to be undefined nowhere (because the case where `b = 0` is explicitly handled and defined), so the function is total: $\{\langle a, b, \texttt{gcd}\ a\ b\rangle \mid a, b \in \mathbb{Z}\}$

(c) Total function: $\{\langle x, |x|\rangle \mid x \in \mathbb{Z}\}$

One can use function graphs to enforce preconditions when implementing the above functions. These are also a concise form of documentation that you might consider putting into a Javadoc or Python docstring. Other programmers (or "future you") will thank you.