BU CS320 Theory Assignment 3

Parser Combinators

# 1. Basic Combinators

We encourage you to first review the parsers.ml file that is posted. This file contains some parsers that were covered in lecture and contains some parsers that are covered in the pre lecture videos. You must review these first before attempting this question.

Describe the purpose (in plain english using few sentences) of the following combinators that are present in parsers.ml

* pure

takes a type of variable(a’) and returns a parser of that type after executing a function writing by another method so if you send It a parser Some(“hello”, []) it will just give you back that same value

* >>=

Is symbol equivalent to the bind method which takes a parser of type a’ and a function that takes a’ to executes whatever function it sent and returns a b’ parser of that answer for example, if you sent it char list parser, with getThreeChars and “hello world” the method returns the first 3 characters of “hello world” in a char list

* <|>

Is the symbol equivalent to the disj method which takes 2 parsers and If the first parser contains some value it will return that tuple but if the first parser is empty It will return the second parsers value tuple so if the first parser is Some(‘h’, []) then it returns the whole tuple else it will return the second parser

* fail

make a parser of type ‘a empty/have no value (None) so if you have a parser of Some(“hello”,[]) and you send it to the fail method it will be equivalent to None

* read

reads the value of char in a parser and if it contains that char the method returns true else false so if Some(‘h’, []) is the parser and the char asked for is ‘e’ it will return false but if the char asked is ‘h’ it will be true

# 2. Derived Combinators

We encourage you to first review the parsers.ml file that is posted. This file contains some parsers that were covered in lecture and contains some parsers that are covered in the pre lecture videos. You must review these first before attempting this question.

Describe the purpose (in plain english using few sentences) of the following combinators that are present in parsers.ml

* many – takes parsers and adds them cons them to gethoer into a list of type a’ parser

runs parser 0 or more times and if it doesn’t succeed it returns a unconsumed function for example, so even if the list is empty it will still return a tuple option like Some([],[])

* many1

runs the parser 1 or more times rather than 0 or more times for example if you were send it “hello world” it would return a tuple option of a char list hello world and an empty list (char list, [])

* char

checks to see if a char sent from a char parser is a given value if it is true its returns the char parse of that char if not false it returns None for example if you sent it ‘h’ and were looking for ‘h’ it will return Some(‘h’, [])

* literal

takes to 2 strings and converts them to char lists and if they have the same values returns a parser called unit of option tuples of all the combinations with the same values so if you gave the method “hello” and checked it with “help” it would return Some(,) of ‘h’, ‘e’ and ‘l’

# 3. Combinator Usage

In the provided parsers.ml file, the type of a parser is given as follows:

type 'a parser = char list -> ('a \* char list) option

Now, instead assume that the type of the parser is:

type 'a parser = char list -> ('a \* char list) list

Rewrite the following combinators:

* <|>

let *disj* (*p1* **:** *'a* **parser**) (*p2* **:** *'a* **parser**) **:** *'a* **parser** **=**

fun *ls* **->**

match p1 ls with

**|** **[**(*x***,** *ls*)**]** **->** **[**(x, ls)**]**

**|** **[]** **->** p2 ls

* >>=

let *bind* (*p* **:** *'a* **parser**) (*q* **:** *'a* **->** *'b* **parser**) **:** *'b* **parser** **=**

fun *ls* **->**

match p ls with

**|** **[]** **->** **[]**

**|**(*h1***,** *rest*)::**[]**

* return – pure

let *pure* (*x* **:** *'a*) **:** *'a* **parser** **=**

fun *ls* **->** **[**(x, ls)**]**

* fail

let *fail* **:** *'a* **parser** **=** fun *ls* **->** **[]**

# 4. Formal Grammars

* Consider the following grammar.

<bool> ::= true | false

<bool\_tree> ::= <bool> | ( <bool\_tree> ^ <bool\_tree> )

Write a leftmost derivation for the following sentences:

( true ^ ( true ^ false ) )

<bool\_tree> ::= <bool\_tree>

::= (<bool\_tree> ^<bool\_tree> )

::= (<bool> ^ <bool\_tree> )

::= (<bool> ^ (<bool\_tree> ^ <bool\_tree> )

::= (<bool> ^ ( <bool> ^ <bool>)

::= (true ^ (<bool> ^ <bool>)

::= ( true ^ ( true ^ <bool>)

::= ( true ^ ( true ^ false)

Write a rightmost derivation for the following sentences:

( ( true ^ true ) ^ ( false ^ false ) )

<bool\_tree> ::= <bool\_tree>

::= (<bool\_tree> ^<bool\_tree> )

::= ((<bool\_tree> ^<bool\_tree>) ^ (<bool\_tree> ^<bool\_tree>))

::= ((<bool\_tree> ^<bool­\_tree>) ^ (<bool> ^<bool>))

::= ((<bool> ^ <bool­>) ^ (<bool> ^ <bool>))

::= ((<bool> ^ <bool­>) ^ (<false> ^ <false>))

::= ((<true> ^ <true­>) ^ (<false> ^ <false>))

* Consider the following grammar.

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<nat> ::= <digit> | <digit><nat>

<opr> ::= + | - | \* | /

<expr> ::= <nat> | ( <opr> <expr> <expr> )

Write a leftmost derivation for the following sentences:

( \* ( + 111 23 ) 9 )

<expr>::=( <opr> <expr> <expr> )

::= (<opr> ( <opr> <expr> <expr> ) <expr>)

::= (<opr> ( <opr> <nat> <nat> ) <nat>)

::= (<opr> ( <opr> <digit><nat> <digit><nat> ) <nat>)

::= (<opr> ( <opr> <digit><digit><nat> <digit><digit> )<nat>)

::= (<opr> ( <opr> <digit><digit><digit> <digit><digit> )<nat>)

::= (<opr> ( <opr> <digit><digit><digit> <digit><digit> )<digit>)

::= ( \* ( <opr> <digit><digit><digit> <digit><digit> )<digit>)

::= ( \* ( + 111 23 )<digit>)

::= ( \* ( + 111 23 ) 9)

Write a rightmost derivation for the following sentences:

( / 10 ( - 11 11 ) )

<expr>::=( <opr> <expr> <expr> )

::=( <opr> <nat> ( <opr> <nat> <nat> ))

::=( <opr> <nat> ( <opr> <digit><nat> <digit><nat> ))

::=( <opr> <nat> ( <opr> <digit><digit> <digit><digit> ))

::=( <opr> <nat> ( - 11 11 ))

::=( <opr> <digit><nat> ( - 11 11 ))

::=( <opr> <digit><digit> ( - 11 11 ))

::=( <opr> 11 ( - 11 11 ))

::=( / 11 ( - 11 11 ))