Advanced Programming
Andrew Black and Tim Sheard

Lecture 6
Regular Expressions
Unit Testing
import HUnit

explode:: String -> [Char]
explode x = x

test1 = testCase
    (assertEqual "explode empty, "
        (explode "")
        []
    )

test2 = testCase
    (assertEqual "Into chars, "
        (explode "abc")
        ['a','b','c']
    )
test3 = testCase( ['\n'] @=? explode "\n")

tests = TestList [test1, test2, test3]
Running tests

Main> runTestTT tests
Cases: 3  Tried: 3  Errors: 0  Failures: 0

bad = TestCase
    (assertEqual "explode reverses?, "
        (explode "ab")
        "ba")

Main> runTestTT bad
### Failure:
explode reverses?,
expected: "ab"
  but got: "ba"
Cases: 1  Tried: 1  Errors: 0  Failures: 1
Hunit flexibility

check s x y = TestCase (assertEqual s y x)

claim:: String -> Bool -> Test
claim s x = check s x True

deny s x = check s x False
Raising a string to a power

toThe :: String -> Int -> [String]

toTheTests = TestList
  [ check "x toThe 0"
      ("x" `toThe` 0) ""
    , check "x toThe 1"
      ("xy" `toThe` 1) "xy"
    , check "x toThe 2"
      ("xy" `toThe` 2) "xyxy"
  ]
Power definition

toThe :: String -> Int -> String
toThe x 0 = ""
toThe x n = x ++ (x `toThe` (n-1))

Main> runTestTT toTheTests
Cases: 3  Tried: 3  Errors: 0  Failures: 0
Regular Expressions

• Let $\Sigma$ be a set of symbols

• Then
  
  – $\epsilon$ is a RE
  
  – $s$ is a RE where $s \in \Sigma$
  
  – $a \cdot b$ is a RE, if $a$ and $b$ are RE
  
  – $a \mid b$ is a RE, if $a$ and $b$ are RE
  
  – $a^*$ is a RE, if $a$ is an RE
data RE
  = Empty
  | Union RE RE
  | Concat RE RE
  | Star RE
  | C Char

val rel1 = Concat (Union (C '+')
  (Union (C '-')
    Empty))
  (Concat (C 'D')
    (Star (C 'D'))))
alpha = Union (C 'a')
    (Union (C 'b') (C 'c'))
digit = Union (C '0')
    (Union (C '1') (C '2'))
key = Union (string "if")
    (Union (string "then")
        (string "else"))
punc = (C ',',)
ident = Concat alpha
    (Star (Union alpha digit))
number = Concat digit (Star digit)
lexer = Union ident (Union number punc)
We can print RE

instance Show RE where
    show Empty = "#"
    show (C x) = [x]
    show (Union x y) = "("++showU x++"++"++showU y++")"
        where showU (Union x y) = show x++"++"++showU y
            showU x = show x
    show (Concat x y) = show x++show y
    show (Star (x@(Concat _ _))) = "("++show x++")*"
    show (Star (x@(Union _ _))) = "("++show x++")*"
    show (Star x) = show x++"*"

Main> show lexer
"((a+b+c)((a+b+c+0+1+2))++(0+1+2)((0+1+2))++,)"
Meaning of a regular expression

• The meaning of a regular expression is a set of strings.
• Sometimes the set is infinite.

• abc(d|e)f = { “abcdf”, “abcef” }
• a* = { “”, “a”, “aa”, “aaa”, “aaaa”, … }
Unit testing \texttt{meaning1}

\begin{verbatim}
check s x y = TestCase (assertEqual s x y)

\texttt{meaning1} :: \texttt{RE} \rightarrow \texttt{[String]}
\texttt{meaning1} = error "No definition for \texttt{meaning1} yet"

\texttt{meaning1Tests} = TestList
\begin{array}{l}
\text{[ check "empty" (meaning1 Empty) []}
, \text{ check "singleton" (meaning1 (C 'a')) ["a"]}
, \text{ check "concat"}
\qquad (meaning1 (Concat (C 'a') (C 'b')))) ["ab"]
, \text{ check "star"}
\qquad (meaning1 (Star (C 'a'))) ["a","aa","aaa"]
, \text{ check "union"}
\qquad (meaning1 (Union (C 'a') (C 'b'))) ["a","b"]
\end{array}
\end{verbatim}
Test driven development

tests = [meaning1Tests ]
test = runTestTT (TestList tests)

Main> test
Cases: 5 Tried: 0 Errors: 0 Failures: 0
Program error: No definition for meaning1 yet
meaning1 :: RE -> [String]
meaning1 Empty = ["""]
meaning1 (Union x y) =
    meaning1 x ++ meaning1 y
meaning1 (C c) = [[c]]
meaning1 (Concat x y) =
    [ as++bs
      | as <- meaning1 x,
      , bs <- meaning1 y ]
meaning1 (Star r) =
    concat [ x `toThe` 3
             | x <- (meaning1 r) ]
Main> runTestTT meaning1Tests
### Failure in: 0
empty
expected: []
    but got: ["""]
### Failure in: 3
star
expected: ["","a","aa","aaa"]
    but got: ["aaa"]
Cases: 5   Tried: 5   Errors: 0   Failures: 2
We wrote a bad test case

```yaml
### Failure in: 0
empty
expected: []
  but got: [""

check "empty" (meaning1 Empty) []

• Should have been

check "empty" (meaning1 Empty) [ "" ]
```
## Failure in: 3

```haskell
star

expected: ["","a","aa","aaa"]
  but got: ["aaa"]

We forgot to add (x `to the` 1) (x `to the` 2) etc.

meaning1 (Star r) =
    concat [ x `toThe` 3
            | x <- (meaning1 r) ]
```
meaning1 :: RE -> [String]
meaning1 (Star r) =
    concat [ x `toThe` i
             | x <- (meaning1 r)
             , i <- [1 .. 3] ]

Main> runTestTT meaning1Tests
### Failure in: 3
star
expected: ["","a","aa","aaa"]
but got: ["a","aa","aaa"]
Cases: 5 Tried: 5 Errors: 0 Failures: 1
meaning1 :: RE -> [String]
meaning1 Empty = ["""]
meaning1 (Union x y) = meaning1 x ++ meaning1 y
meaning1 (C c) = [[c]]
meaning1 (Concat x y) = [ as++bs | as <- meaning1 x, bs <- meaning1 y ]
meaning1 (Star r) = concat [ x `toThe` i |
| x <- (meaning1 r) , i <- [0 .. 3] ]

Main> test
Cases: 3  Tried: 3  Errors: 0  Failures: 0
I thought I was done ..

Main> (meaning1 (Union (C 'a') (C 'b')))
["a","b"]
Main> (meaning1 (Union (C 'b') (C 'a')))
["b","a"]

So add one more test

check "union commutes"
   (meaning1 (Union (C 'b') (C 'a')))
   (meaning1 (Union (C 'a') (C 'b')))
import List(nub,sort)

norm:: [String] -> [String]
norm x = nub(sort x)

meaning1 :: RE -> [String]
meaning1 Empty = ["""]
meaning1 (Union x y) = norm(meaning1 x ++ meaning1 y)
meaning1 (C c) = [[c]]
meaning1 (Concat x y) =
    norm [ as++bs | as <- meaning1 x, bs <- meaning1 y ]
meaning1 (Star r) = norm
    [ x `toThe` i
    | x <- (meaning1 r)
    , i <- [0 .. 3] ]
Main> meaning1 (Star alpha)
["","a","aa","aaa","b","bb","bbb","c","cc","ccc"]

Main> meaning1 ident
["a","a0","a00","a000","a1","a11","a111","a2","a22","a222","aa ","abb","abbb","ac","acc","accc","b","b0","b00","b000","b1","b 22","b222","ba","baa","baaa","bb","bbb","bbbb","bc","bcc","bcc "c000","c1","c11","c111","c2","c22","c222","ca","caa","caaa"," cc","ccc","ccccc"]

Main> meaning1 lexer
["","0","00","000","0000","01","011","0111","02","022","0222" 000","11","111","1111","12","122","1222","2","20","200","2000 " ,"22","222","2222","a","a0","a00","a000","a1","a11","a111","a2 ","aaa","aaaa","ab","abb","abbb","ac","acc","accc","b","b0","b 11","b111","b2","b22","b222","ba","baa","baaa","bb","bbb","bbb c","c","c0","c00","c000","c1","c11","c111","c2","c22","c222"," cb","cbb","cbbb","cc","ccc","ccccc"]
A second Meaning function

meaning2 :: RE -> String -> Bool
meaning2 Empty "" = True

meaning2Tests = TestList
  [  claim "empty2" (meaning2 Empty "")
  ,  deny "Not empty2" (meaning2 Empty "a")
  ,  claim "singleton2" (meaning2 (C 'a') "a")
  ,  deny "Not singleton2" (meaning2 (C 'a') "b")
  ,  claim "concat2"
      (meaning2 (Concat (C 'a') (C 'b')) "ab")
  ,  deny "Not concat2"
      (meaning2 (Concat (C 'a') (C 'b')) "cd")
  ,  claim "0 star2" (meaning2 (Star (C 'a')) "")
  ,  claim "1 star2" (meaning2 (Star (C 'a')) "a")
  ,  claim "2 star2" (meaning2 (Star (C 'a')) "aa")
  ,  claim "3 star2" (meaning2 (Star (C 'a')) "aaa")
  ,  deny "Not star2" (meaning2 (Star (C 'a')) "c")
  ,  claim "union2"
      (meaning2 (Union (C 'a') (C 'b')) "a")
  ,  claim "union commutes2"
      (meaning2 (Union (C 'b') (C 'a')) "b")
]
meaning2 :: RE -> String -> Bool
meaning2 Empty "" = True
meaning2 Empty x = False
meaning2 (C c) [d] = c==d
meaning2 (C c) x = False
meaning2 (Union x y) s =
    meaning2 x s || meaning2 y s
meaning2 (Concat x y) s = any id
    [ meaning2 x (take n s) &&
      meaning2 y (drop n s)
    | n <- [0 .. length s] ]
meaning2 :: RE -> String -> Bool
meaning2 Empty "" = True
meaning2 Empty x = False
meaning2 (C c) [d] = c==d
meaning2 (C c) x = False
meaning2 (Union x y) s =
    meaning2 x s || meaning2 y s
meaning2 (Concat x y) s = all id
    [ meaning2 x (take n s) &&
        meaning2 y (drop n s)
    | n <- [0 .. length s] ]

Main> test
### Failure in: 0:4
concat2
expected: True
but got: False
Cases: 13 Tried: 6 Errors: 0 Failures: 1
Program error: pattern match failure: meaning2 (RE_Star (RE_C 'a')) []

Main> test
### Failure in: 0:4
concat2
expected: True
but got: False
Cases: 13 Tried: 6 Errors: 0 Failures: 1
Program error: pattern match failure: meaning2 (RE_Star (RE_C 'a')) []
meaning2 :: RE -> String -> Bool
meaning2 Empty "" = True
meaning2 Empty x = False
meaning2 (C c) [d] = c==d
meaning2 (C c) x = False
meaning2 (Union x y) s =
    meaning2 x s || meaning2 y s
meaning2 (Concat x y) s = any id [ meaning2 x (take n s) &&
    meaning2 y (drop n s)
    | n <- [0 .. length s] ]
meaning2 (Star x) "" = True
meaning2 (Star x) s = any id [ meaning2 Empty s
    , meaning2 x s
    , meaning2 (Concat x x) s
    , meaning2 (Concat x (Concat x x)) s ]
Cross Testing

- We have two functions that are supposed to compute the same thing. Perhaps we can test this.

crossTest =
  \[\text{claim "cross test"}
  \quad\text{(all id (map (meaning2 ident)
    \quad\text{(meaning1 ident))})}\]

Main> runTestTT crossTest
Cases: 1  Tried: 1  Errors: 0  Failures: 0
Break meaning2

meaning2 (Star x) s = any id
    [ meaning2 Empty s
      , meaning2 x s
      -- , meaning2 (Concat x x) s
      , meaning2 (Concat x (Concat x x)) s ]

Main> runTestTT crossTest
### Failure:
cross test
expected: True
  but got: False
Cases: 1  Tried: 1  Errors: 0  Failures: 1

• Rather un enlightening
Automatic Test generation

genCross re x =
  claim (show x ++ " in " ++ show re)
  (meaning2 re x)

crossTest2 =
  TestList (map (genCross ident)
            (meaning1 ident))

Main> runTestTT crossTest2
### Failure in: 2
"a00" in (a+b+c)((a+b+c+0+1+2))*
expected: True
  but got: False
### Failure in: 5
"all" in (a+b+c)((a+b+c+0+1+2))*
expected: True
  but got: False

Given a RE and a value generates a test.

Generate one big test with many cases.

Many failures (some not shown) but each explains what failed.
Fixing our deliberate error.

Main> runTestTT crossTest2
Cases: 57  Tried: 57  Errors: 0  Failures: 0
Maximal munch

• Another meaning for regular expressions is the longest string possible (at each step)

\[
\text{maxmunch::RE} \to \text{String} \to \text{Maybe(String,String)} \\
\text{maxmunch Empty x = Just("",x)}
\]
First some tests

munchTests = TestList
    [ check "munch empty"
        (maxmunch Empty "abc")
        (Just("","abc"))
    , check "munch ident"
        (maxmunch ident "abc,"
        (Just("abc","",""))
    , check "munch longer"
        (maxmunch (Union ident key) "thenx,123"
        (Just("thenx","","123"))
    , check "munch fails"
        (maxmunch (C 'x') "abc") Nothing
    ]

Main> runTestTT munchTests
Cases: 4  Tried: 1  Errors: 0  Failures: 0
Program error: pattern match failure:
    maxmunch ident "abc,"

\[\text{longer } \text{Nothing Nothing} = \text{Nothing} \]
\[\text{longer } (\text{Just } x) \text{ Nothing} = \text{Just } x \]
\[\text{longer } \text{Nothing } (\text{Just } x) = \text{Just } x \]
\[\text{longer } (\text{Just}(x,xs)) \ (\text{Just}(y,ys)) = \]
\[\text{if } x_n \geq y_n \text{ then } \text{Just}(x,xs) \]
\[\text{else } \text{Just}(y,ys) \]
\text{where } x_n = \text{length } x
\text{yn} = \text{length } y
seqm f g cs =
  case f cs of
    Nothing -> Nothing
    Just(pre,post) ->
      case g post of
        Nothing -> Just(pre,post)
        Just(pre2,post2) ->
          Just(pre++pre2,post2)
maxmunch :: RE -> String ->
      Maybe(String,String)
maxmunch Empty cs = Just("",cs)
maxmunch (C x) (c:cs) =
    if x==c then Just([c],cs) else Nothing
maxmunch (Concat x y) cs =
    seqm (maxmunch x) (maxmunch y) cs
maxmunch (Union x y) cs =
    longer (maxmunch x cs) (maxmunch y cs)
maxmunch (Star x) cs = repeat cs
    where repeat cs =
        seqm (maxmunch x) repeat cs
Main> maxmunch ident "thenx,123"
Nothing

Main> runTestTT munchTests
### Failure in: 2
munch longer
expected: Just ("thenx",",123")
but got: Just ("then","x,123")
Cases: 4 Tried: 4 Errors: 0 Failures: 1

Main> maxmunch ident "thenx,123"
Nothing
Main> ident
(a+b+c)((a+b+c+0+1+2))
oneOf [] = Empty
oneOf [x] = C x
oneOf (x:xs) = Union (C x) (oneOf xs)

alpha2 = oneOf "then1s"
ident2 = Concat alpha2
       (Star (Union alpha2 digit))

check "munch longer"
       (maxmunch (Union ident2 key)"thent,123")
       (Just("thent","",123"))

Main> runTestTT munchTests
Cases: 4 Tried: 4 Errors: 0 Failures: 0
nullable

nullable :: RE -> Bool
nullable Empty = True

nullableTests = TestList
    [ claim "Empty null" (nullable Empty)
    , claim "null a*" (nullable (Star (C 'a')))
    , claim "null <e>*" (nullable (Star Empty))
    , deny "not null a" (nullable (C 'a'))
    , claim "null (<e> | a)(b <e>)"
    , deny "not null <e>a" (nullable (Concat Empty (C 'a'))) 
    , claim "null <e><e>" (nullable (Concat Empty Empty)) 
    ]

Main> test
Cases: 29  Tried: 23  Errors: 0  Failures: 0
Program error: pattern match failure:
    nullable (RE_Star (RE_C 'a'))
nullable :: RE -> Bool
nullable Empty = True
nullable (C _) = False
nullable (Star _) = True
nullable (Concat x y) =
  nullable x && nullable y
nullable (Union x y) =
  nullable x || nullable y

Main> :r
Main> test
Cases: 29  Tried: 29  Errors: 0  Failures: 0
Conclusions

• Regular expressions are a language
• We can treat them as programs
  – Every program can have multiple meanings
    • meaning1
    • meaning2
    • maxmuch
  – Programs are both code and data
    • We can execute them
    • We can print them
    • We can analyze them
      – nullable
Conclusions 2

• The HUInt framework is flexible
• We can define our own notion of tests
  – check
  – claim
  – deny
• Tests are data structures
  – We can write our own flexible test generators
    • crossCheck
Assignment #6

• For homework, using the RE framework write the functions match1 and consumes1 using test driven development framework.

\[ \text{match1} :: \text{RE} \rightarrow \text{Char} \rightarrow \text{Bool} \]

\((\text{match1} \ x \ c)\) returns True if \(c\) is the first character of a string generated by \(x\). This should be an exact answer and not limit \((\text{Star} \ x)\) to three iterations.

Then write

\[ \text{consumes1} :: \text{RE} \rightarrow \text{Char} \rightarrow \text{Maybe} \ \text{RE} \]

\((\text{consumes1} \ x \ c)\) returns \((\text{Just} \ y)\) if \((c:cs)\) is a string generated by \(x\), and \(cs\) is a string generated by \(y\). Again, this should be an exact answer and not limit \((\text{Star} \ x)\) to three iterations.