50.054 Parsec

ISTD, SUTD

Learning Outcomes

- Implement parsers using Parser Combinator

Naive Top Down parsing

- Base case: `symbols` is `[]`
  1. the parse tree for the current rule `N ::= RHS` must have been constructed and we just return it.
- Recursive case: `symbols` is `symbol::symbols'`

1. if the leading `symbol` is a terminal
   1.1 if the input token list is `tok::toks` and `tok` matches with `symbol`, construct the leaf of the parse tree. Move on to the next token/symbol, i.e. `toks` and `symbols'`.
   1.2 otherwise signals a failure
2. if the leading `symbol` is a non-terminal `M`
   2.1 if the input token list is `Nil` and a rule `M::= RHS2` exists in the grammar
      2.1.1 if `RHS2` accepts empty tokens, construct the empty parse tree leaf w.r.t `M`. Move on to the next symbol, i.e. parsing `Nil` with `symbols`.
      2.1.2 if `RHS2` does not accept empty tokens, signals a failure.
   2.2 if the input token list is `tok::toks`, **pick an alternative `M::= RHS'`**, apply recursion with the rule `M::= RHS'` and `tok::toks`. Keep trying until one alternative succeeds in parsing `tok::toks`.
   2.3 otherwise signal a failure.

Example

```
(1) T ::= xx
(2) T ::= yx
```

```
enum LToken {
    case XTok
    case YTok
}

enum T {
    case XX
    case YX
}
```

Naive implementation

```
enum Result[A] {
  case Failed(msg:String)
  case Ok(v:A)
}

def item(toks:List[LToken]):Result[(LToken, List[Token])] = toks match {
  case Nil => Failed("item() is called with an empty input")
  case (t::ts) => Ok((t, ts))
}

def sat(toks:List[LToken])(p:LToken => Boolean):Result[(LToken, List[Token])] = toks match {
  case Nil => Failed("sat() is called with an empty input")
  case (t::ts) if p(t) => Ok((t, ts))
  case (t::ts)         => Failed("""sat() is called with an input that
   does not satisfy the input predicate.""")
}
```

Naive implementation

```
def parseT(toks:List[LToken]):Result[T] = {
    parseXX(toks) match {
        case Failed(_) => parseYX(toks) match {
            case Failed(_) => Failed("parse error")
            case Ok((yx, Nil))    => Ok(yx)
            case Ok((yx, toks2)) => Failed("some tokens haven't been parse
        }
        case Ok((t,Nil))  => Ok(t)
        case Ok((t,_))    => Failed("some tokens haven't been parsed.")
    }
}
```

Naive implementation

```
def parseXX(toks:List[LToken]):Result[(T, List[LToken])] = {
    sat(toks)( t => t match {
        case XTok => true
        case _ => false
    }) match {
        case Failed(err) => Failed(err)
        case Ok((x1,toks1)) => {
            sat(toks1)( s => s match {
                case XTok => true
                case _ => false
            }) match {
                case Failed(err) => Failed(err)
                case Ok((x2,toks2)) => Ok((XX,toks2))
            }
        }
    }
}
```

Naive implementation

```
def parseYX(toks:List[LToken]):Result[(T, List[LToken])] = {
    sat(toks)( t => t match {
        case YTok => true
        case _ => false
    }) match {
        case Failed(err) => Failed(err)
        case Ok((y1,toks1)) => {
            sat(toks1)( s => s match {
                case XTok => true
                case _ => false
            }) match {
                case Failed(err) => Failed(err)
                case Ok((x2,toks2)) => Ok((YX,toks2))
            }
        }
    }
}
```

Issues

- Code duplicates - `sat()` is called twice with the same lambda in `parseXX`
- Boiler plate codes, `parseXX` and `parseYX`

Let's fix it using Monad

```scala
case class Parser[T, A](p: List[T] => Result[(A, List[T])]) {
    def map[B](f: A => B): Parser[T, B] = this match
        case Parser(p) =>
            Parser(toks =>
                p(toks) match
                    case Failed(err)    => Failed(err)
                    case Ok((a, toks1)) => Ok((f(a), toks1)))

    def flatMap[B](f: A => Parser[T, B]): Parser[T, B] = this match {
        case Parser(p) =>
            Parser(toks =>
                p(toks) match
                    case Failed(err) => Failed(err)
                    case Ok((a, toks1)) =>
                        f(a) match
                            case Parser(pb) => pb(toks1))
    }
}

def run[T, A](parser: Parser[T, A])
    (toks: List[T]): Result[(A, List[T])] = parser match {
    case Parser(p) => p(toks)
}
```

Let's fix it using Monad

```scala
type ParserM = [T] =>> [A] =>> Parser[T, A]

given parsecMonadError[T]: MonadError[ParserM[T], Error] =
    new MonadError[ParserM[T], Error] {
        override def pure[A](a: A): Parser[T, A] =
            Parser(cs => Ok((a, cs)))
        override def bind[A, B](
            fa: Parser[T, A]
        )(f: A => Parser[T, B]): Parser[T, B] = fa.flatMap(f)
        override def raiseError[A](e: Error): Parser[T, A] =
            Parser(toks => Failed(e))
        override def handleErrorWith[A](
            fa: Parser[T, A]
        )(f: Error => Parser[T, A]): Parser[T, A] = fa match {
            case Parser(p) =>
                Parser(toks =>
                    p(toks) match {
                        case Failed(err) => run(f(err))(toks)
                        case Ok(v)       => Ok(v)
                    }
                )
        }
    }
```

Let's fix it using Monad

```
def choice[T, A](p: Parser[T, A])(q: Parser[T, A])(using
    m: MonadError[ParserM[T], Error]
): Parser[T, A] = m.handleErrorWith(p)(e => q)

def item[T]: Parser[T, T] =
    Parser(toks => {
        toks match {
            case Nil =>
                Failed(s"item() is called with an empty token stream")
            case (c :: cs) => Ok((c, cs))
        }
    })

def sat[T](p: T => Boolean, err:String=""): Parser[T, T] = Parser(toks =>
    toks match {
        case Nil => Failed(s"sat() is called with an empty token stream. ${err}")
        case (c :: cs) if p(c) => Ok((c, cs))
        case (c :: cs) =>
            Failed(s"sat() is called with a unsatisfied predicate with ${c}. ${err}")
    }
)
```

Let's fix it using Monad

```
def parseT:Parser[LToken,T] = choice(parseXX)(parseYX)

def parseXX:Parser[LToken,T] = parseX.flatMap(
    x => parseX.map( x => XX )
)

def parseYX:Parser[LToken,T] = parseY.flatMap(
    y => parseX.map( x => YX )
)

def parseX:Parser[LToken, LToken] = sat(t => t match {
    case XTok => true
    case _ => false
})

def parseY:Parser[LToken, LToken] = sat(t => t match {
    case YTok => true
    case _ => false
})
```

Let's fix it using Monad

```
def parseT:Parser[LToken, T] = choice(parseXX)(parseYX)

def parseXX:Parser[LToken, T] = for {
    _ <- parseX
    _ <- parseX
} yield XX

def parseYX:Parser[LToken, T] = for {
    _ <- parseY
    _ <- parseX
} yield YX

def parseX:Parser[LToken, LToken] = sat(t => t match {
    case XTok => true
    case _ => false
})

def parseY:Parser[LToken, LToken] = sat(t => t match {
    case YTok => true
    case _ => false
})
```

More Combinators from the Parsec library

```
def optional[T, A](pa: Parser[T, A]): Parser[T, Either[Unit, A]] = {
        val p1: Parser[T, Either[Unit, A]] = for (a <- pa) yield (Right(a))
        val p2: Parser[T, Either[Unit, A]] = Parser(toks => Ok((Left(()), toks)))
        choice(p1)(p2)
    }

// one or more
def many1[T, A](p: Parser[T, A]): Parser[T, List[A]] = for {
    a <- p
    as <- many(p)
} yield (a :: as)

// zero or more
def many[T, A](p: Parser[T, A]): Parser[T, List[A]] = ... // omitted, refer to the cohort problem
```

Back to the MathExp parsing

```
enum LToken { // lexical Tokens
    case IntTok(v: Int)
    case PlusTok
    case AsterixTok
}

enum Exp {
    case TermExp(t:Term)
    case PlusExp(t:Term, e:Exp)
}

enum Term {
    case FactorTerm(f:Factor)
    case MultTerm(t:Term, f:Factor)
}

case class Factor(v:Int)
```

```
<<grammar 4>>
E::= T + E
E::= T
T::= T * F
T::= F
F::= i
```

Back to the MathExp parsing

```
def parseExp:Parser[LToken, Exp] =
    choice(parsePlusExp)(parseTermExp)

def parsePlusExp:Parser[LToken, Exp] = for {
    t <- parseTerm
    plus <- parsePlusTok
    e <- parseExp
} yield PlusExp(t, e)

def parseTermExp:Parser[LToken, Exp] = for {
    t <- parseTerm
} yield TermExp(t)
```

```
<<grammar 4>>
E::= T + E
E::= T
T::= T * F
T::= F
F::= i
```

Parsing a Term

```
def parseTerm:Parser[LToken, Term] = ???
```

```
T::= T * F  // left recursive!!
T::= F
F::= i
```

Handling Left Recursion

```
def parseTerm:Parser[LToken, Term] = for {
    tle <- parseTermLE
} yield fromTermLE(tle)

case class TermLE(f:Factor, tp:TermLEP)

enum TermLEP {
    case MultTermLEP(f:Factor, tp:TermLEP)
    case Eps
}

def parseTermLE:Parser[LToken, TermLE] = ???
def fromTermLE(tle:TermLE):Term = ???
```

```
T::= T * F  // left recursive!!
T::= F
F::= i
```

le-grammar

```
T  ::= FT'
T' ::= *FT'
T' ::= epsilon
```

Handling Left Recursion

```
def parseTermLE:Parser[LToken, TermLE] = for {
    f <- parseFactor
    tp <- parseTermP
} yield TermLE(f, tp)

def parseTermP:Parser[LToken, TermLEP] = for {
    omt <- optional(parseMultTermP)
} yield { omt match {
    case Left(_) => Eps
    case Right(t) => t
}}

def parseMultTermP:Parser[LToken, TermLEP] = for {
    asterix <- parseAsterixTok
    f <- parseFactor
    tp <- parseTermP
} yield MultTermLEP(f, tp)
```

```
T::= T * F  // left recursive!!
T::= F
F::= i
```

le-grammar

```
T  ::= FT'
T' ::= *FT'
T' ::= epsilon
```

Handling Left Recursion

```
def parseFactor:Parser[LToken, Factor] = for {
    i <- parseIntTok
    f <- someOrFail(i)( itok => itok match {
        case IntTok(v) => Some(Factor(v))
        case _         => None
    })("""
    parseFactor() fail: expect to parse
    an integer token but it is not an integer.""")
} yield f

def parsePlusTok:Parser[LToken, LToken] = sat ((x:LToken) => x match {
    case PlusTok => true
    case _       => false
}, "Expecting a + symbol")

def parseAsterixTok:Parser[LToken, LToken] = ... // omitted

def parseIntTok:Parser[LToken, LToken] = ... // omitted

// apply f to a to extract b, if the result is None, signal failure
def someOrFail[T, A, B](a:A)( f:A=>Option[B])(err:Error):Parser[T, B] = Parser( toks => f(a) match
    case Some(b) => Ok((b, toks))
    case None => Failed(err)
)
```

```
T::= T * F  // left recursive!!
T::= F
F::= i
```

le-grammar

```
T  ::= FT'
T' ::= *FT'
T' ::= epsilon
```

Handling Left Recursion

```
case class TermLE(f:Factor, tp:TermLEP)

enum TermLEP {
    case MultTermLEP(f:Factor, tp:TermLEP)
    case Eps
}

enum Term {
    case FactorTerm(f:Factor)
    case MultTerm(t:Term, f:Factor)
}

def fromTermLE(t:TermLE):Term = t match {
    case TermLE(f, tep) =>
        fromTermLEP(FactorTerm(f))(tep)
}
def fromTermLEP(t1:Term)(tp1:TermLEP):Term =
    tp1 match {
        case Eps => t1
        case MultTermLEP(f2, tp2) => {
            val t2 = MultTerm(t1, f2)
            fromTermLEP(t2)(tp2)
        }
    }
```

```
T::= T * F  // left recursive!!
T::= F
F::= i
```

le-grammar

```
T  ::= FT'
T' ::= *FT'
T' ::= epsilon
    T
   / \
  f   T'
     /|\
    * f T'
       /|\
      * f T'
          |
         eps
```

and the parse tree of Term is

```
        T
       /|\
      T * f
     /| \
    T * f
    |
    f
```

Exercise time

work on cohort exercises 1, 2 and 4.

Exploiting LL(1) with Parsec

- For `LL(1)` grammar we can pick the right choice w/o backtracking
- But with the current version of parser combinator we are always backtracking! We can modify it such that
  - Backtracking by default, no-backtracking with explicit combinator
  - No Backtracking by default, backtracking with explicit combinator (Our option)
    - on demand backtracking

Parsec w/ on-demand backtracking

```
enum Progress[+A] { // progress tracking constructors
    case Consumed(value: A)
    case Empty(value: A)
}
```

Parsec w/ on-demand backtracking

```scala
case class Parser[T, A](p: List[T] => Progress[Result[(A, List[T])]]) {
    def map[B](f: A => B): Parser[T, B] = this match {
        case Parser(p) =>
            Parser(toks =>
                p(toks) match {
                    case Empty(Failed(err))    => Empty(Failed(err))
                    case Empty(Ok((a, toks1))) => Empty(Ok((f(a), toks1)))
                    case Consumed(Failed(err)) => Consumed(Failed(err))
                    case Consumed(Ok((a, toks1))) =>
                        Consumed(Ok((f(a), toks1)))
                }
            )
    }
}
```

Parsec w/ on-demand backtracking

```scala
case class Parser[T, A](p: List[T] => Progress[Result[(A, List[T])]]) {
    def flatMap[B](f: A => Parser[T, B]): Parser[T, B] = this match
        case Parser(p) =>
            Parser(toks =>
                p(toks) match
                    case Consumed(v) =>
                        lazy val cont = v match
                            case Failed(err) => Failed(err)
                            case Ok((a, toks1)) =>
                                f(a) match
                                    case Parser(pb) =>
                                        pb(toks1) match
                                            case Consumed(x) => x
                                            case Empty(x)    => x
                        Consumed(cont)
                    case Empty(v) =>
                        v match
                            case Failed(err) => Empty(Failed(err))
                            case Ok((a, toks1)) =>
                                f(a) match
                                    case Parser(pb) => pb(toks1)
            )
}
```

Parsec w/ on-demand backtracking

```
given parsecMonadError[T]: MonadError[ParserM[T], Error] =
    new MonadError[ParserM[T], Error] {
        override def handleErrorWith[A](fa: Parser[T, A]
        )(f: Error => Parser[T, A]): Parser[T, A] = fa match
            case Parser(p) =>
                Parser(toks =>
                    p(toks) match
                        case Empty(Failed(err)) =>
                            // only backtrack when it is empty
                                f(err) match
                                    case Parser(p2) => p2(toks)
                        case Empty(Ok(v)) =>
                            // LL(1) parser will also attempt to
                            // look at f if fa does not consume anything
                                f("faked error") match
                                    case Parser(p2) =>
                                        p2(toks) match
                                            case Empty(_) => Empty(Ok(v))
                                                // if f also fail, we report the error from fa
                                            case consumed => consumed
                        case Consumed(v) => Consumed(v)
                )
    }
```

Parsec w/ on-demand backtracking

```scala
def choice[T, A](p: Parser[T, A])(q: Parser[T, A])(using
    m: MonadError[ParserM[T], Error]
): Parser[T, A] = m.handleErrorWith(p)(e => q)

def item[T]: Parser[T, T] =
    Parser(toks => {
        toks match
            case Nil => Empty(Failed(s"item() is called with an empty token stream"))
            case (c :: cs) => Consumed(Ok((c, cs)))
    })

def sat[T](p: T => Boolean, err:String=""): Parser[T, T] = Parser(
    toks => {
        toks match {
            case Nil => Empty(Failed(s"sat() is called with an empty token stream. ${err}"))
            case (c :: cs) if p(c) => Consumed(Ok((c, cs)))
            case (c :: cs) => Empty(Failed(
                    s"sat() is called with a unsatisfied predicate with ${c}. ${err}"
                ))
        }
    }
)
```

Nearly no change except the insertion of `Empty` and `Consumed`.

Parsec w/ on-demand backtracking

No change!

```
def parseT:Parser[LToken,T] = choice(parseXX)(parseYX) // no backtracking!

def parseXX:Parser[LToken,T] = parseX.flatMap(
    x => parseX.map( x => XX )
)

def parseYX:Parser[LToken,T] = parseY.flatMap(
    y => parseX.map( x => YX )
)

def parseX:Parser[LToken, LToken] = sat(t => t match {
    case XTok => true
    case _ => false
})

def parseY:Parser[LToken, LToken] = sat(t => t match {
    case YTok => true
    case _ => false
})
```

Parsec w/ on-demand backtracking

```
run(parseT)(List(XTok, XTok)) ---> // defn of run
parserT match { case Parser(p) => p(List(XTok, XTok))} ---> // defn of parserT
choice(parseXX)(parseYX) match { case Parser(p) => p(List(XTok, XTok))} --->  // defn of choice
handleErrorWith(parseXX)(e => parseYX) match { case Parser(p) => p(List(XTok, XTok))} ---> *
Consumed(lazy Ok((XTok, List(XTok))) match
        case Failed(err) => Failed(err)
        case Ok((a, toks1)) =>
            x => parseX.map( x => XX ) (a) match
                case Parser(pb) =>
                    pb(toks1) match
                        case Consumed(x) => x
                        case Empty(x)    => x
)
```

refer to the annex for the rest of the derivation.

Parsec w/ backtracking

```
run(parseT)(List(XTok, XTok)) ---> // defn of run
parserT match { case Parser(p) => p(List(XTok, XTok))} ---> // defn of parserT
choice(parseXX)(parseYX) match { case Parser(p) => p(List(XTok, XTok))} --->  // defn of choice
handleErrorWith(parseXX)(e => parseYX) match { case Parser(p) => p(List(XTok, XTok))} ---> *

(Ok((XTok, List(XTok))) match
    case Failed(err) => Failed(err)
    case Ok((a, toks1)) =>
        (x => parseX.map(x => XX)(a)) match
            case Parser(pb) => pb(toks1)
) match {                                                     // the backtracking code
    case Failed(err) => run((e => parseYX)(err))(toks)        // is chaininng up
    case Ok(v)       => Ok(v)                                 // still
}
```

refer to the annex for the rest of the derivation.

When we need on-demand backtracking

```
enum U {
    case XX
    case XY
}

def parseU:Parser[LToken,U] = choice(parseXX)(parseXY) // no backtracking!

def parseXX:Parser[LToken,U] = parseX.flatMap(
    x => parseX.map( x => XX )
)

def parseXY:Parser[LToken,U] = parseY.flatMap(
    y => parseX.map( x => XY )
)

run(parseU)(List(XTok, YTok)) // fails, as it commit to parseXX
// when the first XTok is consumed
```

When we need on-demand backtracking

```
enum U {
    case XX
    case XY
}

def parseU:Parser[LToken,U] = choice(attempt(parseXX))(parseXY) // no back

run(parseU)(List(XTok, YTok)) // succeeds, as it will backtrack to parseXY

// explicit try and backtrack if fails
def attempt[T, A](p: Parser[T, A]): Parser[T, A] =
    Parser(toks =>
        run(p)(toks) match {
            case Consumed(Failed(err)) => Empty(Failed(err))
            case otherwise             => otherwise
        }
    )
```

Summary

- Parsec with default backtracking
- Parsec with on-demand backtracking