Parser Combinators

KS Sreeram
Tachyon Technologies

Introduction

In this lecture, we will use higher-order procedures to greatly simplify the calculator program. Higher-order parser procedures are also called *Parser Combinators*. We will see that parsers created with parser combinators are structurally similar to the grammar of the language being parsed. This makes parser combinators a powerful and generic tool for building arbitrary parsers.

What are Higher-Order Procedures?

Procedures operate on data. Higher-order procedures operate on procedures.

If the same computation is performed in multiple places, operating on different values, then procedures can be used for abstraction.

If the same control-flow is used in multiple places with different computations, then higher-order procedures can be used for abstraction.

Consider the following two procedures:

```
def doubleList(a) :
    result = []
    for x in a :
        result.append(x + x)
    return result

def squareList(a) :
    result = []
    for x in a :
        result.append(x * x)
    return result
```

We see that 'doubleList' and 'squareList' have the same control flow, with different computations being performed. We can write a higher-order procedure which factors out the common control flow, and accepts the actual computation being performed as a procedure parameter.

```
def map(f, a) :
    result = []
    for x in a :
        result.append(f(x))
    return result
```

Here, 'f' is a procedure value which is applied to every element of the input list. The output list consists of the all the values returned by 'f'. Since, 'map' accepts a procedure as a parameter, it is a higher-order procedure.

Now, 'doubleList' and 'squareList' can written using 'map' as follows:

```
def doubleList(a) :
    def double(x) :
        return x + x
    return map(double, a)

def squareList(a) :
    def square(x) :
        return x * x
    return map(square, a)
```

The 'lambda' keyword can be used to create unnamed procedures as follows.

```
def doubleList(a) :
    return map(lambda x: x+x, a)

def squareList(a) :
    return map(lambda x: x*x, a)
```

The 'map' higher-order procedure captures a commonly occurring control-flow pattern where one list is transformed into another.

Let's look at examples where we reduce a list into a single value.

```
def sum(a) :
    result = 0
    for x in a :
        result = result + x
    return result

def product(a) :
    result = 1
    for x in a :
        result = result * x
    return result
```

We see that 'sum' and 'product' have the same control flow with different initial values and different computations being performed. The following higher-order procedure factors out this pattern.

```
def reduce(f, a, initial) :
    result = initial
    for x in a :
        result = f(result, x)
    return result
```

We can now write 'sum' and 'product' as follows. We use 'lambda' to conveniently create anonymous procedures.

```
def sum(a) :
    return reduce(lambda x,y:x+y, a, 0)

def product(a) :
    return reduce(lambda x,y:x*y, a, 1)
```

Note that in many of the above definitions we have used nested procedures. One important feature of nested procedures is that they remember the values of variables from outer scopes. The following example illustrates this.

```
def makeAdder(delta) :
    def adder(x) :
        return x + delta
    return adder

f = makeAdder(10)
g = makeAdder(100)

print f(2)  # prints 12
print g(2)  # prints 102
```

This completes our quick tour of higher-order procedures!

Identifying Control Flow Patterns

As we've seen before, repeatedly occurring control flow patterns can be factored out using higher-order procedures. Let's look for such patterns in the calculator program.

We see that procedures 'parseDigit', 'parseOpeningParenthesis', 'parseClosingParenthesis', 'parseStar', and 'parsePlusOrMinus' are similar. All of these procedures consume one character by calling 'next', and check if a specific condition is true of that character. The parser succeeds if the condition is true of that character.

program0.py

```
def parseDigit() :
    c = next()
    code = ord(c)
    if (code < 48) or (code > 57) :
        error('invalid digit: ' + c)
    return code - 48

def parseOpeningParenthesis() :
    c = next()
    if c != '(' :
        error('expected (, but received: ' + c)

def parseClosingParenthesis() :
    c = next()
    if c != ')' :
        error('expected ), but received: ' + c)
```

```
def parseStar() :
    c = next()
    if c != '*' :
        error('expected *, but received: ' + c)

def parsePlusOrMinus() :
    c = next()
    if (c != '+') and (c != '-') :
        error('expected one of + -, but received: ' + c)
    return c
```

We also see that the procedures 'parseNumeral', 'parseTerm', and 'parseExpr' have the same control flow. They have the pattern of repeated application a parser.

program0.py

```
def parseNumeral() :
    result = parseDigit()
    while True :
        p = save()
        try :
            d = parseDigit()
            result = result*10 + d
        except ParserError :
            restore(p)
            break
    return result

def parseTerm() :
    a = parseFactor()
    while True :
        p = save()
        try :
            parseStar()
            b = parseFactor()
            a = a * b
        except ParserError :
            restore(p)
            break
    return a

def parseExpr() :
    a = parseTerm()
    while True :
        p = save()
        try :
            op = parsePlusOrMinus()
            b = parseTerm()
            a = applyPlusOrMinus(op, a, b)
        except ParserError :
            restore(p)
            break
```

```
    return a
```

Combinators - *condition*, *zeroOrMore*, and *oneOrMore*

We factor out the recurring patterns above with higher-order procedures. Such higher-order parser procedures are called parser combinators.

The combinator 'condition' accepts the condition procedure as a parameter, and returns a parser procedure which consumes a single character. The given condition is applied on the character, and if the condition is met, then the parser succeeds and returns the character. If the condition is not met, then an error is flagged.

```
def condition(f) :
    def proc() :
        x = next()
        if not f(x) :
            error("parser error")
        return x
    return proc
```

The combinator 'zeroOrMore' accepts a parser procedure as input, and returns a parser procedure which applies the input parser zero or more times. The input parser is applied as many times as possible until failure. Upon failure, the returned parser recovers by restoring the current position to the value just before the failed call to the input parser. All the parse results of the input parser are accumulated in a list and returned as the result of the parser.

```
def zeroOrMore(parser) :
    def proc() :
        result = []
        while True :
            p = save()
            try :
                x = parser()
                result.append(x)
            except ParserError :
                restore(p)
                break
        return result
    return proc
```

The combinator 'oneOrMore' accepts a parser procedure as input, and returns a parser procedure which applies the input parser atleast once. If the first application of the input parser fails, then the returned parser fails too. After the first application of the input parser, we need to repeatedly apply the input parser until failure. Repeated application until failure can be performed by creating a parser with 'zeroOMore'.

```
def oneOrMore(parser) :
    rest = zeroOrMore(parser)
    def proc() :
        return [parser()] + rest()
```

```
    return proc
```

Using these combinators, we can write a digit and numeral parser as follows:

program1.py

```
def isDigit(x) :
    return (x >= '0') and (x <= '9')

digit = condition(isDigit)
numeral = oneOrMore(digit)
```

Combinators - Adding the *modifier* option

Here is a transcript of running the previous program.

```
$ python program1.py

> 1
['1']

> 123
['1', '2', '3']
```

We would like to obtain the numeric value corresponding to the numerals rather than just character lists. We cannot directly add this conversion logic to the combinators. This is because we would like to use these combinators in many other cases besides just parsing numerals.

We can instead update the combinators to accept an additional procedure as input which modifies the result as needed. This retains the generality of the combinators while adding the required feature.

```
def condition(f, modifier=lambda x : x) :
    def proc() :
        x = next()
        if not f(x) :
            error("parser error")
        return modifier(x)
    return proc
```

We have made 'modifier' an optional parameter whose default value is the do-nothing identity function.

We change the 'oneOrMore' combinator similarly.

```
def oneOrMore(parser, modifier=lambda x : x) :
    rest = zeroOrMore(parser)
    def proc() :
        return modifier([parser()] + rest())
    return proc
```

We can now update the 'digit' and 'numeral' parsers to return numeric values.

program2.py

```
def isDigit(x) : return (x >= '0') and (x <= '9')
def digitToNum(x) : return ord(x) - ord('0')

def combineDigits(a) :
    return reduce(lambda n,d: n*10+d, a, 0)

digit = condition(isDigit, digitToNum)
numeral = oneOrMore(digit, combineDigits)
```

Calculator Using Combinators

Proceeding as above, we can convert the entire calculator program to use combinators.

program3.py

```
def isDigit(x) : return (x >= '0') and (x <= '9')
def digitToNum(x) : return ord(x) - ord('0')

def combineDigits(a) :
    return reduce((lambda x, y : x*10+y), a, 0)

def applyOp(a, op, b) :
    if op == '*' : return a * b
    elif op == '+' : return a + b
    elif op == '-' : return a - b
    assert False

def mergeTail(x) :
    return reduce((lambda u, v : applyOp(u, v[0], v[1])), x[1], x[0])

digit = condition(isDigit, modifier=digitToNum)
numeral = oneOrMore(digit, modifier=combineDigits)
expr = lambda : expr2()
parenthesized = sequence([char('('), expr, char(')')], lambda x : x[1])
factor = choice([numeral, parenthesized])
termTail = zeroOrMore(sequence([char('*'), factor]))
term = sequence([factor, termTail], modifier=mergeTail)
plusOrMinus = choice([char('+'), char('-')])
exprTail = zeroOrMore(sequence([plusOrMinus, term]))
expr2 = sequence([term, exprTail], modifier=mergeTail)
```

We see that the parser is structurally similar to the grammar of the calculator.

```
numeral       <- digit+
parenthesized <- '(' expr ')'
factor        <- numeral | parenthesized
```

```
term          <- factor ('*' factor)*
plusOrMinus   <- '+' | '-'
expr          <- term (plusOrMinus term)*
```

Exercise 1

In *program3.py*, what is the purpose of the definition:

```
    expr = lambda : expr2()
```

Exercise 2

Implement parser combinators using object-oriented programming rather than using higher-order procedures.

Exercise 3

Improve error reporting in parser combinators.