In [8]:
```
# Styling notebook
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[8]:

**Recursive Descent (RD) Samples**

Some of these are very basic and lack a couple of details that might depend on the OS, input/output conventions, etc.

In general, these examples use (comparbaly short) strings to hold the source code, so to run them in a file system environment you might have to work on the details or (preferably) the Scanner - which is in the Scanner notebook.

The third (last) version uses the Scanner that should be present. Note that the "include" does not require the notebook file (ipynb) but rather the Python source file (py) which can easily be created from the notebook via File --> Download as --> Python (.py). *You really need Scanner.py to run the last example!*

**Example 1:** So without much ado, here's the first trivial example. You might want to watch the explanation in the lecture to figure out how this works.

Note that using a CFG + RD to parse this is overkill - it's regular, so a FA/RegEx is sufficient!

**EBNF:**

`sequence = letter { letter }`

`letter = (a | b)`

In [9]:
```
class Parser1 :
    def __init__(self, source) :
        self.source = source + '$'
        self.sourcePos = 0

    # This assert FALSE is the way to abort a running Python program
    # It will trigger an AssertionError exception which you can with try/except
    # --> the "parse" function
    def error(self,message) :
        print(message)
        assert False

    # sequence = letter { letter }
    def sequence(self) :
        print("sequence",self.sourcePos,self.lookahead)
        self.letter()
        while self.lookahead in {'a','b'} :
            self.letter()

        # Also a standard twist:
        # When you're done, check if you're really at EOF
        if self.lookahead != '$' :
            self.error('$ expected, found ' + str(self.lookahead))

    # Letter = (a | b)
    def letter(self) :
        print("letter",self.sourcePos,self.lookahead)
        if self.lookahead in {'a','b'} :
            self.sourcePos = self.sourcePos + 1
            self.lookahead = self.source[self.sourcePos]
        else :
            self.error('a or b expected, found ' + str(self.lookahead))

    def parse(self) :
        try:
            self.sourcePos = 0
            self.lookahead = self.source[self.sourcePos]
            self.sequence()
            print('Success!')
        except AssertionError :
            print("Aborted!")
```

In [10]:

```
p = Parser1("aabaaba")
p.parse()
```

```
sequence 0 a
letter 0 a
letter 1 a
letter 2 b
letter 3 a
letter 4 a
letter 5 b
letter 6 a
Success!
```

**Example 2:** This is a little more sophisticated. Since there's no semantics involved (yet), we just assume that "x" is the only identifier and 5 is the only numerical constant.

**EBNF:**

```
expression = term  { ["+" | "-"] term} ;
term       = factor  { ["*"|"/"] factor} ;
factor     = ident | number | "("  expression  ")" ;
ident    = "x" ;
constant    = "5" ;
```

In [11]:

```
class Parser2 :
    def __init__(self, source) :
        self.source = source + '$'
        self.sourcePos = 0

    # Little more elaborate Scanner/Error interface

    # Same trick as always
    def error(self,message) :
        print('Error at position', self.sourcePos)
        print(message)
        assert False

    def scan(self) :
        self.sourcePos = self.sourcePos + 1
        self.lookahead = self.source[self.sourcePos]

    def expect(self,symbolSet) :
        if self.lookahead in symbolSet :
            if self.lookahead != '$' : self.scan()
        else :
            self.error(str(symbolSet) + ' expected, found ' + str(self.lookahead))

    # expression = term  { ["+" | "-"] term} ;
    def expression(self) :
        self.term()
        while self.lookahead in {'+','-'} :
            self.scan()
            self.term()

    # term = factor  { ["*"|"/"] factor} ;
    def term(self) :
        self.factor()
        while self.lookahead in {'*','/'} :
            self.scan()
            self.factor()


    # factor = ident | number | "("  expression  ")" ;
    def factor(self) :
        if self.lookahead == 'x' :
            self.ident()
            return
        if self.lookahead == '5' :
            self.number()
            return
        if self.lookahead == "(" :
            self.scan()
            self.expression()
            self.expect({')'})
            return
        self.error('{x,5,(} expected, found ' + str(self.lookahead))

    # ident = "x" ;
    def ident(self) :
        self.expect({'x'})

    # number = "5" ;
    def number(self) :
        self.expect({'5'})

    def parse(self) :
        try:
            # Prologue: Scan first symbol before descent
            self.sourcePos = 0
            self.lookahead = self.source[self.sourcePos]
            # Call start symbol
            self.expression()
            # Epilogue: Check for EOF
            self.expect({'$'})
            print('Success!')
        except AssertionError :
            print("Aborted!")
```

In [12]:
```
p = Parser2("5*(5-x)*x-5*5")
p.parse()
```

```
Success!
```

**Example 3:** This is a little more advanced, but still a terribly simplified EBNF-style grammar that does some basic PL constructs: assignments, conditions, loops and expressions. There are statement sequences but right now no blocks (exercise). This is why the conditions and loops look so weird.

This "LVal" is just an abstraction for anything you can assign to: identifiers, struct components, array elements, you name it. Here we're using just identifiers

We'll use this a little later to study IR code generation.

*Again: Watch it! You need Scanner.py to run this!*

AExpr = Term (+ |- Term)* Term = Factor (* | / Factor)* Factor = (AExpr) | id | num Stmts = Stmt (;Stmt)* Stmt = Assign | Cond | Loop Cond = if ( AExpr ) Stmt Loop = while ( AExpr ) Stmt Assign = LVal = AExpr LVal = id

In [13]:
```
from Scanner import Scanner

class Parser3 :
    def __init__(self, source) :
        self.source = source
        self.scanner = Scanner(source)
        self.start = self.Stmts
        self.currentToken = None
        self.currentValue = None

    def error(self,message) :
        print("Error: " + message)
        assert False

    def scan(self) :
        self.currentToken, self.currentValue = self.scanner.nextToken()

    def expect(self,symbolSet) :
        if self.currentToken in symbolSet :
            if self.currentToken != 'eoI' : self.scan()
        else :
            self.error(str(symbolSet) + ' expected, found ' + str(self.currentToken))

    def parse(self) :
        try:
            # Prologue: Scan first symbol before descent
            self.scan()
            # Call start symbol
            self.start()
            # Epilogue: Check for EOF
            self.expect({'eoI'})
            print('Success!')
        except AssertionError :
            print("Aborted!")

    # Stmts  = Stmt (;Stmt)*
    def Stmts(self) :
        self.Stmt()
        while (self.currentToken == 'semicolon') :
            self.scan()
            self.Stmt()

    # Stmt   = Assign | Cond | Loop
    # We computed the FIRST sets and found they were disjoint. YAY!!
    def Stmt(self) :
        if self.currentToken == 'identifier' :
            self.Assign()
        elif (self.currentToken == 'ifSym'):
            self.Cond()
        elif (self.currentToken == 'whileSym'):
            self.Loop()
        else:
            self.error("id, if or while expected")

    # Assign = LVal = AExpr
    def Assign(self) :
        self.LVal()
        self.expect({'assignSym'})
        self.AExpr()

    # Cond   = if ( AExpr ) Stmt
    def Cond(self) :
        self.scan() # Watch it - we're still standing on the "if", so move on!
        self.expect({'lParen'})
        self.AExpr()
        self.expect({'rParen'})
        self.Stmt()

    # Loop   = while ( AExpr ) Stmt
    def Loop(self) :
        self.scan() # You're getting it, I'm sure!
        self.expect({'lParen'})
        self.AExpr()
        self.expect({'rParen'})
        self.Stmt()

    # LVal = id
    def LVal(self) :
        self.scan() # We know it's an id, so just scan and go back
        # Note that this may backfire if the FIRST(LVal) has more than one symbol
        # Factor deals with this, so check the difference

    # AExpr  = Term (+ |- Term)*
    def AExpr(self) :
        self.Term()
        while self.currentToken in {'plusSym', 'minusSym',} :
            self.scan() # Watch it!!
            self.Term()

    # Term   = Factor (* | / Factor)*
    def Term(self) :
        self.Factor()
        while self.currentToken in {'timesSym', 'divSym',} :
            self.scan()
            self.Factor()

    # Factor = (AExpr) | id | num
    # Here you have to look at the specific symbol
    # Reason FIRST(Factor) = {(,id,num}
    # Now the whole thing is quite weird for several reasons
    # 1. The current token hasn't been verified, so we have to do that
    #    This self.expect takes care of this, BUT...
    # 2. If that's successful, expect scans forward
    #    Meaning after the expect the current token is gone --> Gotta save it on "toke
    #    Meaning (atm) identifiers and numConstants have nothing to do - that will cha
    #    Meaning I wouldn't even have to write the tests, but I do since something's g
    #    That's why we use this "pass"
    def Factor(self) :
        token = self.currentToken
        self.expect({'lParen', 'identifier', 'numConstant'})
        if token == 'lParen' :
            self.AExpr()
            self.expect({'rParen'});
        elif self.currentToken == 'identifier' :
            pass
        elif self.currentToken == 'numConstant' :
            pass
```

In [14]:

```
#s = "xyz = 12; xyz=xyz - 1 + 5"
s = "a = 1; b = a + 2; if (a*b) a = b+1; while (a) b = a+b"
# buggy:
# s = "a = )1; b = a + 2; if (a*b) a = b+1; while (a) b = a+b"
#s = "a = 1 b = a + 2; if (a*b) a = b+1; while (a) b = a+b"
p = Parser3(s)
p.parse()
```

```
Success!
```