4 Recursive-Descent Parsing

Implementing a Recursive-descent Top-down Parser

1) <S> → <A><C>
2) <A> → 'a' 'b'
3) <C> → 'c'<C>
4) <C> → 'd'

Figure 4.1

`A()` function advances past an A-string (a terminal string derivable from <A>).
`B()` function advances past a B-string (a terminal string derivable from <B>).

Then the `S()` function is given by

```
41 def S():
42    A()
43    C()
```

`S()` advances past a S-string.

```
consume('a')    # advances if current token is ‘a’; error o.w.
```

```
45 def A():
46    consume('a')
47    consume('b')
```

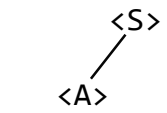
```
49 def C():
50    if token == 'c':
51       # perform actions corresponding to production 3
52       advance()
53       C()                              # recursive call
54    elif token == 'd':
55       # perform action corresponding to production 4
56       advance()
57    else:
58       raise RuntimeError('Expecting c or d')
```

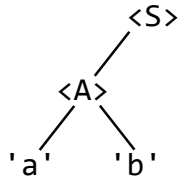
```
# sp.py parser
# Grammar:
#    <S> -> <A><C>
#    <A> -> 'a''b'
#    <C> -> 'c'<C>
#    <C> -> 'd'
import sys   # needed to access command line arg

#global variables
tokenindex = -1
token = ''

def main():
   try:
      parser()         # call the parser
   except RuntimeError as emsg:
      print(emsg)

def advance():
   global tokenindex, token
   tokenindex += 1     # move tokenindex to next token
   # check for null string or end of string
   if len(sys.argv) < 2 or tokenindex >= len(sys.argv[1]):
      token = ''      # signal the end by returning ''
   else:
      token = sys.argv[1][tokenindex]

def consume(expected):
   if expected == token:
      advance()
   else:
      raise RuntimeError('Expecting ' + expected)

def parser():
   advance()   # prime token with first character
   S()         # call function for start symbol
   # test if end of input string
   if token != '':
      print('Garbage following <S>-string')

def S():
   A()
   C()

def A():
   consume('a')
   consume('b')

```

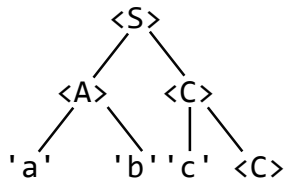
```
49 def C():
50     if token == 'c':
51         # perform actions corresponding to production 3
52         advance()
53         C()
54     elif token == 'd':
55         # perform action corresponding to production 4
56         advance()
57     else:
58         raise RuntimeError('Expecting c or d')
59
60 main()
```

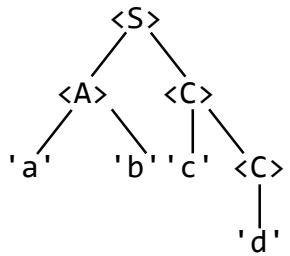
Figure 4.2

Recursive-descent parsing determines parse tree









Trying out sp.py

```
python sp.py abd
python sp.py abccd
python sp.py
python sp.py abdd
```

When Not to Use Recursion

```
<C> → 'c' <C>     (these productions generate c*d)
<C> → 'd'
```

```
50    if token == 'c':
51       # perform actions corresponding to production 3
52       advance()
53       C()
54    elif token == 'd':
55       # perform action corresponding to production 4
56       advance()
57    else:
58       raise RuntimeError('Expecting c or d')
```

Better:

```
<C> → ('c')* 'd'
```

```
1 def C():
2    while token == 'c':
3       advance()              # advance over the c's
4    consume('d')              # check for d at end
```

But use recursion for <X>

```
<X> → 'a' <X> 'b' 'c'
<X> → 'd'
```

```
1 def X():
2    if token == 'a':
3       advance()
4        X()
5       consume('b')
6       consume('c')
7    elif token == 'd':
8       advance()
9    else:
10      raise RuntimeError('Expecting a or d')
```

Using Grammars with the *, +, |, and [] Operators

```
<A> → 'a'* 'b'          (zero or more a's followed by b)
<B> → 'a'+              (one or more a's)
<C> → 'a'('b'|'c')      (a followed by b or c)
<D> → 'a'['b']          (a optionally followed by b)
```

```
1 # <A> → 'a'* 'b'
2 def A():
3    while token == 'a':
4       advance()
5    consume('b')
```

```
1 # <B> → 'a'+
2 def B():
3    consume('a')
4    while token == 'a':
5       advance()
```

```
1 # <C> → 'a'('b'|'c')
2 def C():
3    consume('a')
4    if token == 'b':
5       advance()
6    elif token == 'c':
7       advance()
8    else:
9       raise RuntimeError('Expecting b or c')
```

```
1 # <D> → 'a'['b']
2 def D():
3    consume('a')
4    if token == 'b':
5       advance()
```