5 Predict Sets

Analyzing Grammars for Predictive Recursive-descent Parsing

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

Figure 5.1

```
def A():
   if token == 'a':   # use production 3
      advance()
      consume('b')
   elif token == 'b': # use production 4
      advance()
      consume('b')
   else:
      raise RuntimeError('Expecting a or b')
```

```
def S():
   if token == 'a' or token == 'b':
      A()
   elif token == 'c' or token == 'd':
      C()
   else:
      raise RuntimeError('Expecting a, b, c, or d')
```

Can also tell which production to apply

	Predict sets
1) <S> → <A>	{'a', 'b'}
2) <S> → <C>	{'c', 'd'}
3) <A> → 'a' 'b'	{'a'}
4) <A> → 'b' 'b'	{'b'}
5) <C> → 'c' 'c'	{'c'}
6) <C> → 'd' 'd'	{'d'}

Figure 5.2

Prediction with same left side that are not mutually disjoint

this is a problem

```
                                 Predict sets
1) <S> → <A>                     {'a', 'b'}
2) <S> → <C>                     {'a', 'd'}
3) <A> → 'a' 'b'                 {'a'}
4) <A> → 'b' 'b'                 {'b'}
5) <C> → 'a' 'c'                 {'a'}
6) <C> → 'd' 'd'                 {'d'}
```

Figure 5.3

1. Rewrite the grammar so this problem does not occur.

2. Have the parser look ahead in the token stream to determine which production to apply. For this grammar if a is followed by b, then the parser is positioned at the beginning of an <A>-string, in which case A() should be called. If, on the other hand, a is followed by c, then the parser is positioned at the beginning of a <C>-string, in which case C() should be called.

3. Use the first <S> production. If it does not work (i.e., the parse fails), then backtrack to the current point in the token stream and try the second <S> production.

Predict set for lambda production

1) <S> → <A><B>	{'a','b'}	
2) <S> → <C>	{'c'}	
3) <A> → 'a'	{'a'}	
4) <A> → λ	{'b'}	why 'b'?
5) <B> → 'b'	{'b'}	
6) <C> → 'c' 'd'	{'c'}	

Figure 5.4

```
def S():
   if token == 'a' or token == 'b':
      A()
      B()
   elif token == 'c':
      C()
   else:
      raise RuntimeError('Expecting a, b, or c')

def A():
   if token == 'a':   # test if <A> -> a should be applied
      advance()
   elif token == 'b'  # test if <A> -> lambda should be applied
      pass            # Python stmt that does nothing
   else:
      raise RuntimeError('Expecting a or b')
```

Determining predict sets

<A> → *rightsideofproduction*

1) The predict set for this production includes all the leading terminals in the strings that can be derived from *rightsideofproduction*. We refer to this set as FIRST(*rightsideofproduction*). Suppose the production is

   <A> → <B><C><D>

   Then FIRST(*rightsideofproduction*) = FIRST(<B><C><D>). FIRST(<B><C><D>) includes everything in FIRST(<B>). But if <B> is nullable, then FIRST(<B><C><D>) *also* includes everything in FIRST(<C>). If both <B> and <C> are nullable, then FIRST(<B><C><D>) *also* includes everything in FIRST(<D>).

2) If *rightsideofproduction* is λ or nullable (i.e., it can generate the null string), then the predict set *also* includes the set of terminal symbols that can follow the left side of the production. We refer to the set of terminal symbols that can follow <A> as FOLLOW(<A>).

3) There is always some kind of end-of-input marker at the end of the string we are parsing. Thus, the FOLLOW set of the start symbol *always* contains that marker

4) Whatever follows the left side of a production follows any symbol on the right side that is either rightmost or has only nullables to its right.

Example

	Predict sets
1) <S> → <A><B>	{'a', 'b', EOF}
2) <A> → 'a'	{'a'}
3) <A> → λ	{'b', EOF}
4) <B> → 'b'	{'b'}
5) <B> → λ	{EOF}

Figure 5.6

Productions with a Choice

Consider the following grammar:

predict sets for each production

```
<S> → 'a'(B|C)'d'    {'a'}
<B> → b              {'b'}
<C> → c*             {'c', 'd'}
```

choice

Figure 5.7

The predict set for the <S> production is {'a'}. However, within this production there is a choice—between <B> and <C>. Associated with the <B> choice *is aother predict set*. Similarly, associated with the <C> choice *is a third predict set*. The predict sets for <B> and <C> determine which function—B() or C()—the parser calls after it advances past the initial 'a'.

Here is the corresponding S(), B() and C() functions:

predict set for <S> production
predict set for <B> choice
predict set for <C> choice

```
def S():
   consume('a')
   if token == 'b':
      B()
   elif token in ['c', 'd']:
      C()
   else:
      raise RuntimeError("Expecting 'b', 'c', or 'd'")
   consume('d')

def B():
   advance()  # token is 'b' here so no need to use consume('b')

def C():
   while token == 'c': # structure for a starred item (see page 33)
      advance()
```

Figure 5.8