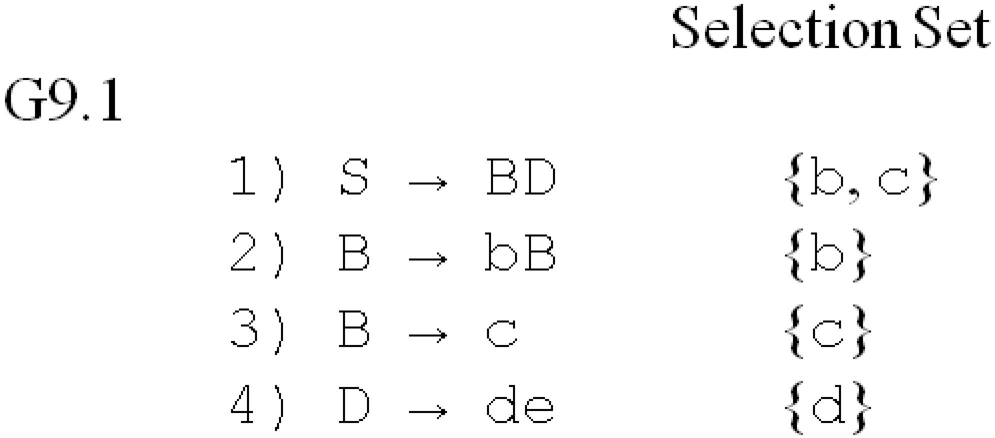
# Chapter 9

## Recursive-descent parsing

# Write a method for each nonterminal



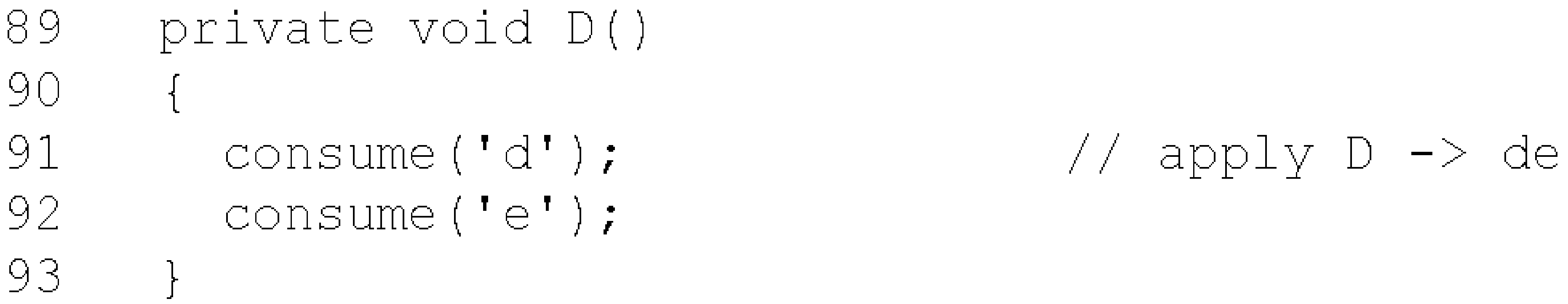
G9.1

| | Selection Set |
|---|---|
| 1) $S \to BD$ | $\{b, c\}$ |
| 2) $B \to bB$ | $\{b\}$ |
| 3) $B \to c$ | $\{c\}$ |
| 4) $D \to de$ | $\{d\}$ |

# B methods advances past B string

```
private void B()
{
  switch(currentToken)
  {
    case 'b':
      consume('b');                 // apply B -> bB
      B();
      break;
    case 'c':
      advance();                    // apply B -> c
      break;
    default:
      throw new RuntimeException("\"b\" or \"c\"");
  }
}
```

# D method advances past D string

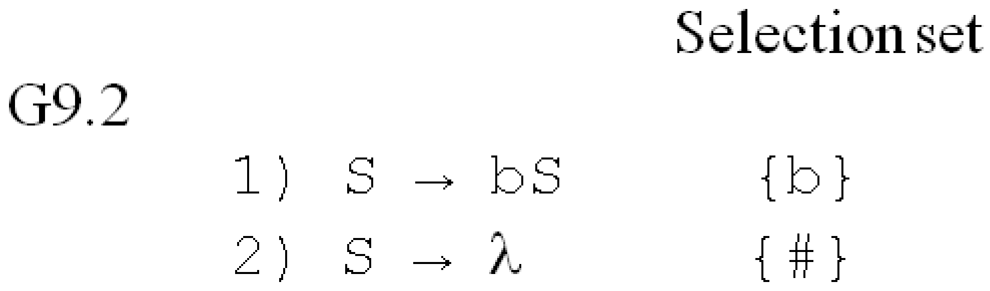


```
private void D()
{
  consume('d');                    // apply D -> de
  consume('e');
}
```

# S method advances past S string

```
private void S()
{
  B();                              // apply S -> BD
  D();
}
```

# Handling lambda productions



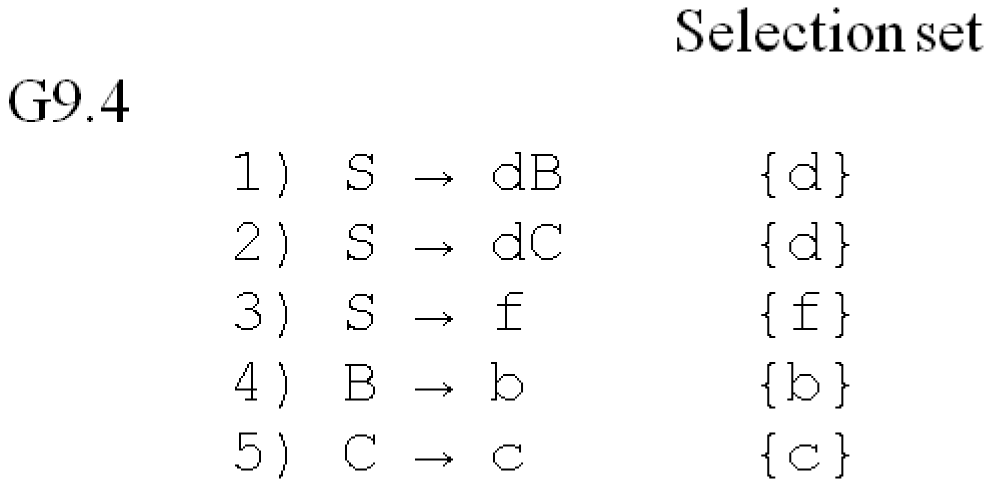
G9.2

|  |  | Selection set |
|---|---|---|
| 1) | S → bS | {b} |
| 2) | S → λ | {#} |

# Use null statement

```
private void S()
{
  switch(currentToken)
  {
    case 'b':       // {b} is selection set for prod 1
      consume('b'); // apply production 1
      S();
      break;
    case '#':       // {#} is selection set for prod 2
      ;             // apply lambda production
      break;
    default:
      throw new RuntimeException("\"b\" or end of input");
  }
}
```

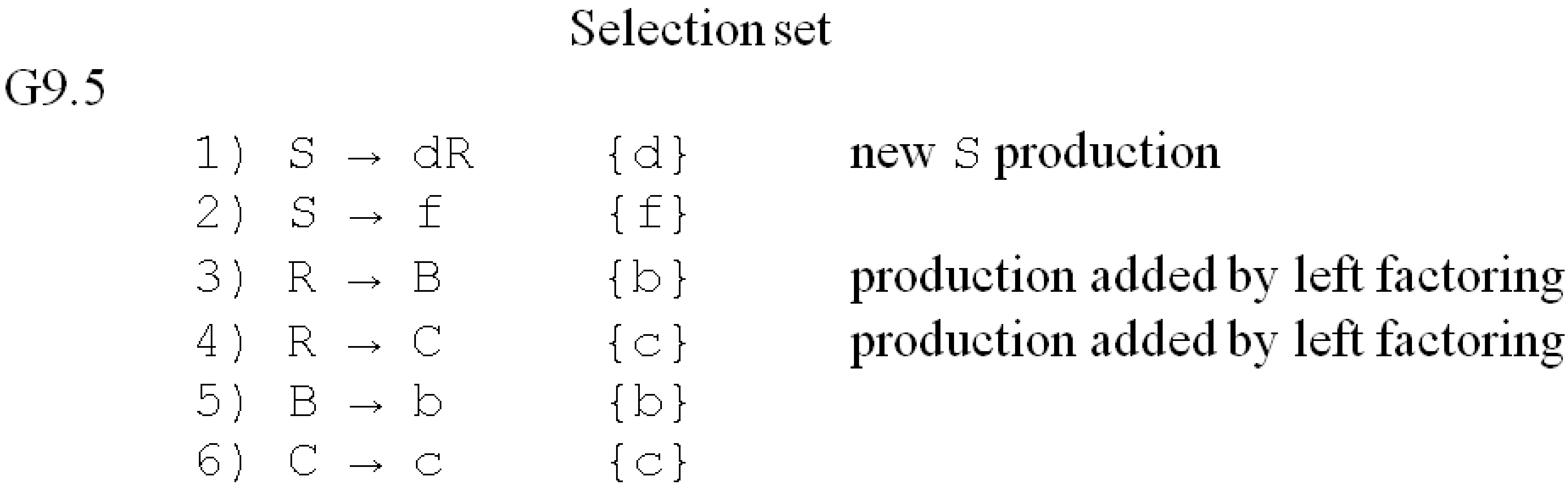
# Left factoring



G9.4

| | Selection set |
|---|---|
| 1) S → dB | {d} |
| 2) S → dC | {d} |
| 3) S → f | {f} |
| 4) B → b | {b} |
| 5) C → c | {c} |

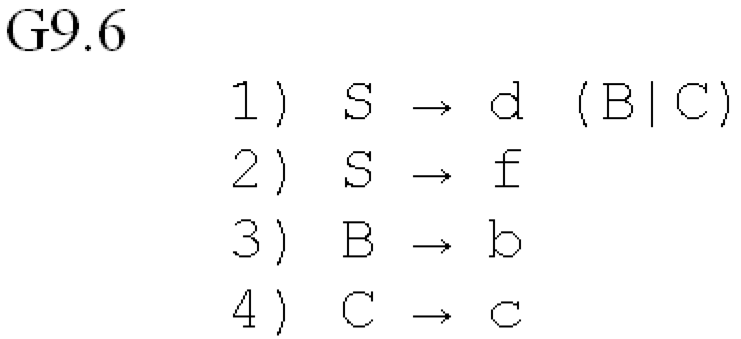
# Equivalent grammar



G9.5

| | Production | Selection set | |
|---|---|---|---|
| 1) | S → dR | {d} | new S production |
| 2) | S → f | {f} | |
| 3) | R → B | {b} | production added by left factoring |
| 4) | R → C | {c} | production added by left factoring |
| 5) | B → b | {b} | |
| 6) | C → c | {c} | |

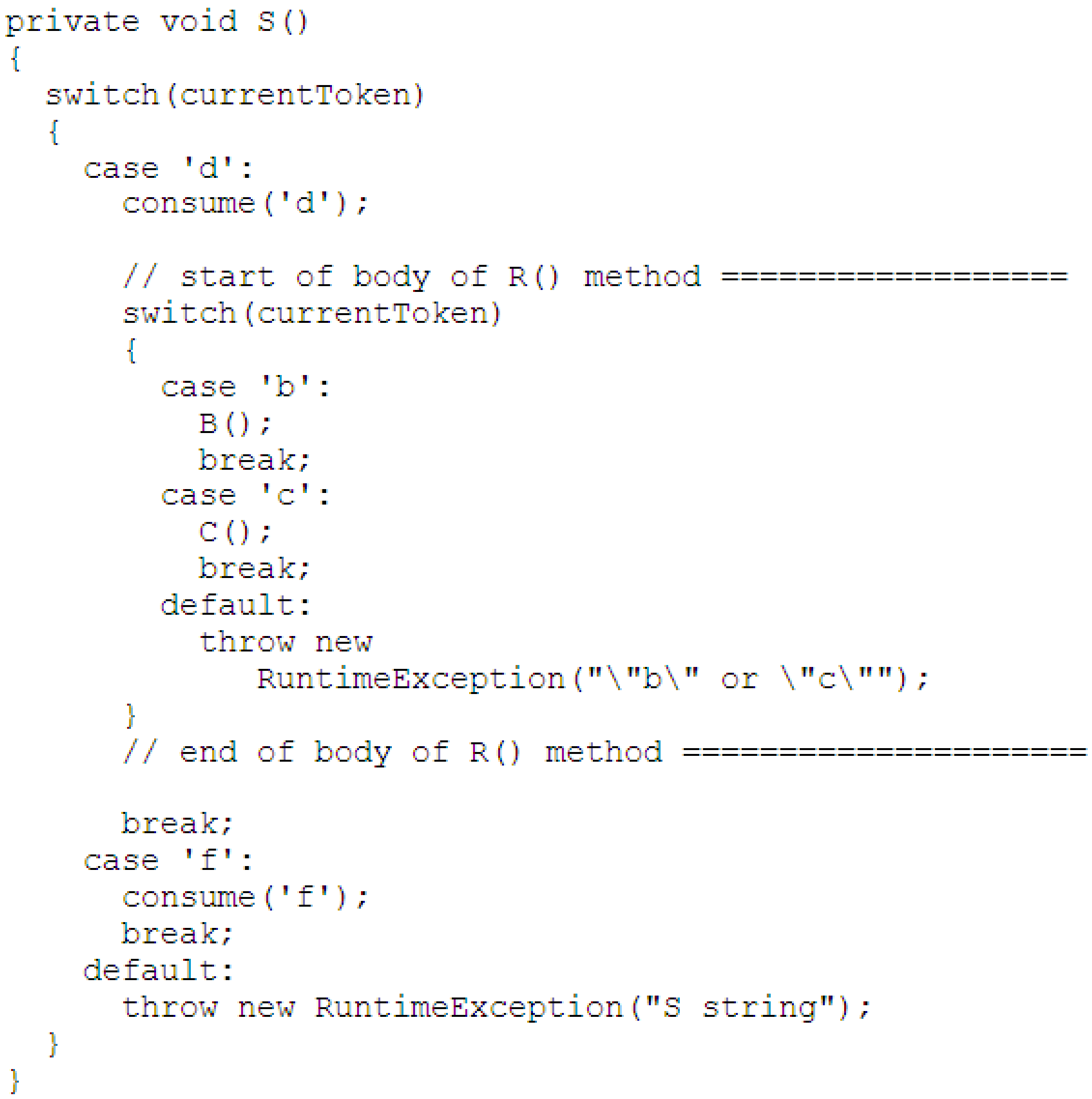
# Equivalent grammar



G9.6

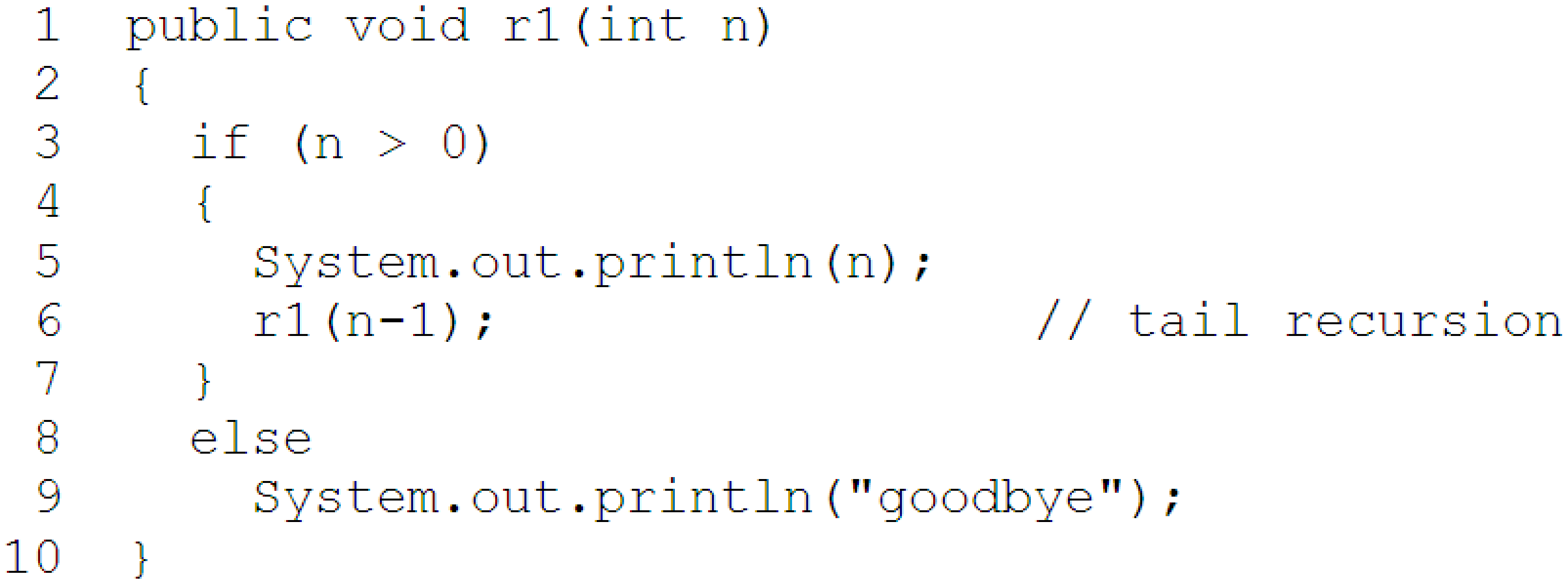
```
1)  S → d (B|C)
2)  S → f
3)  B → b
4)  C → c
```

# Embed R method in S



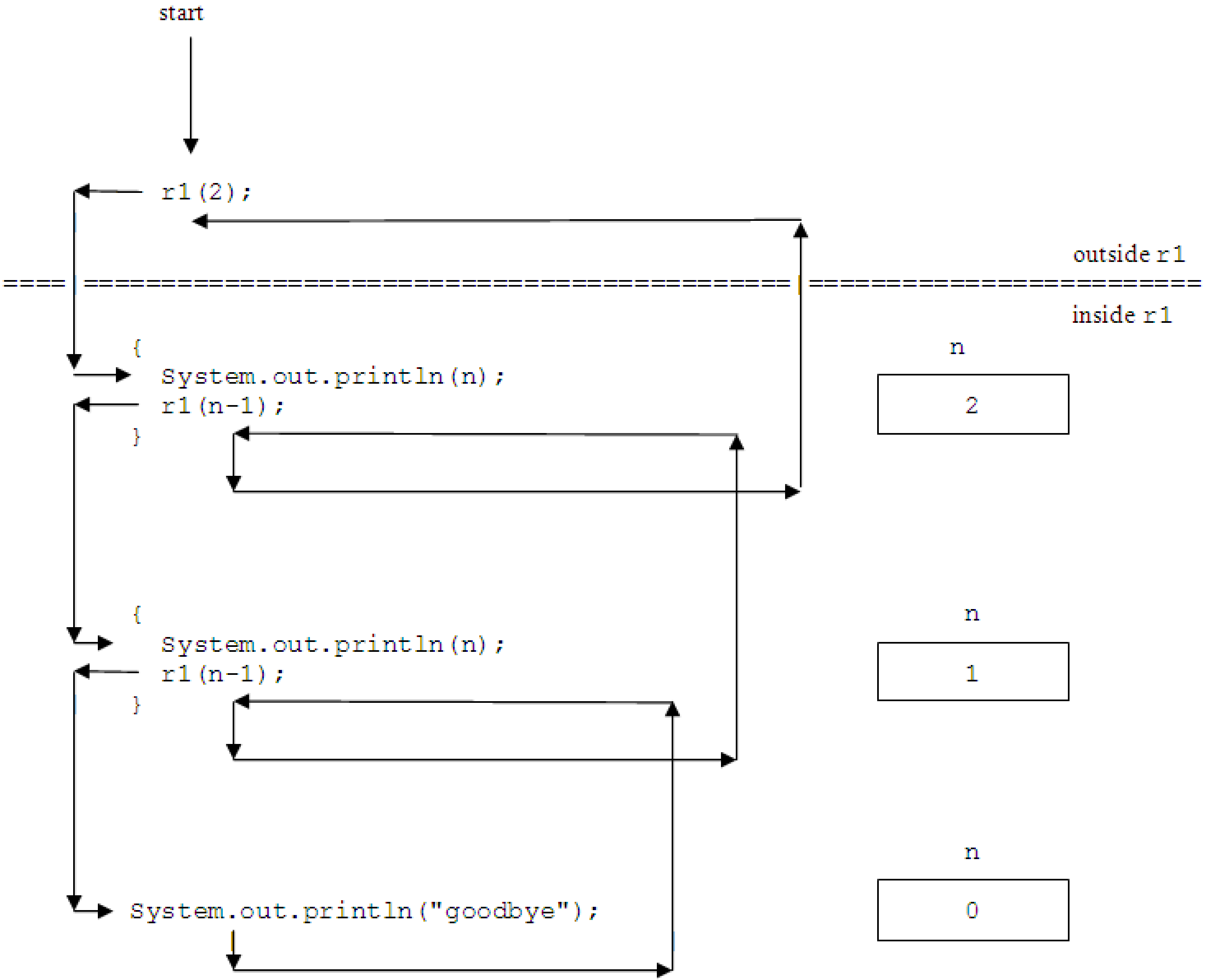
```
private void S()
{
  switch(currentToken)
  {
    case 'd':
      consume('d');

      // start of body of R() method ==================
      switch(currentToken)
      {
        case 'b':
          B();
          break;
        case 'c':
          C();
          break;
        default:
          throw new
             RuntimeException("\"b\" or \"c\"");
      }
      // end of body of R() method =====================

      break;
    case 'f':
      consume('f');
      break;
    default:
      throw new RuntimeException("S string");
  }
}
```

# Tail recursion



```
public void r1(int n)
{
  if (n > 0)
  {
    System.out.println(n);
    r1(n-1);                    // tail recursion
  }
  else
    System.out.println("goodbye");
}
```

# Only returns after recursive call



start

```
r1(2);
```

outside r1

inside r1

```
{
  System.out.println(n);
  r1(n-1);
}
```

n

2

```
{
  System.out.println(n);
  r1(n-1);
}
```

n

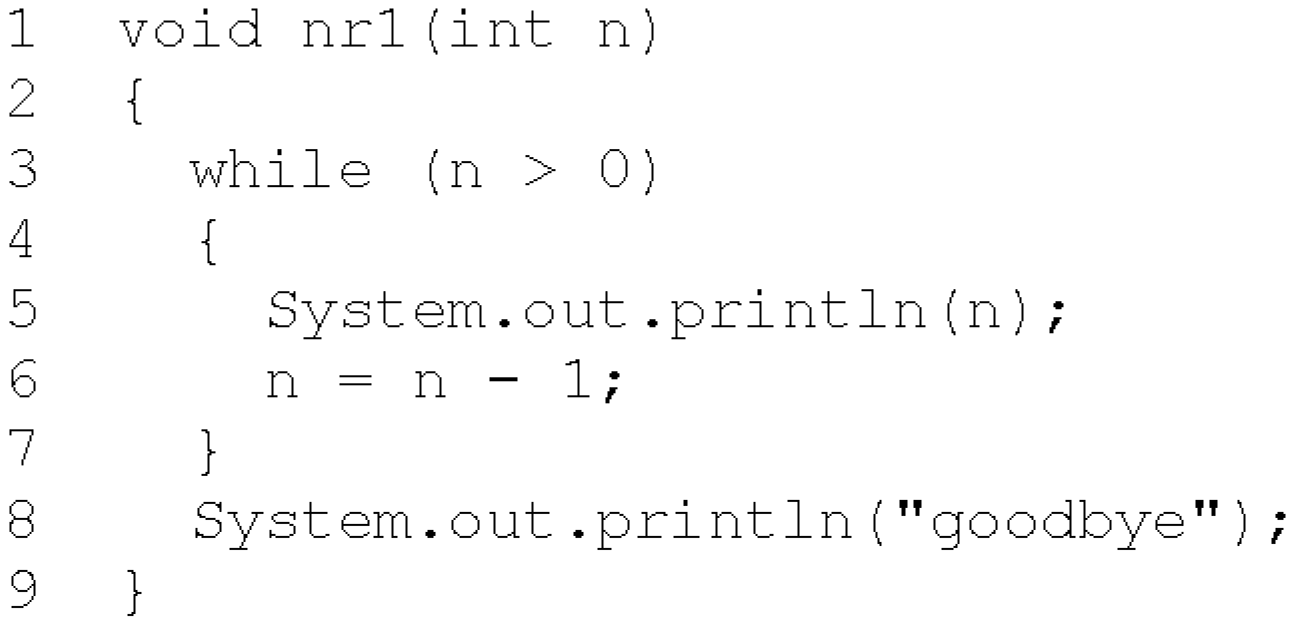
1

```
System.out.println("goodbye");
```

n

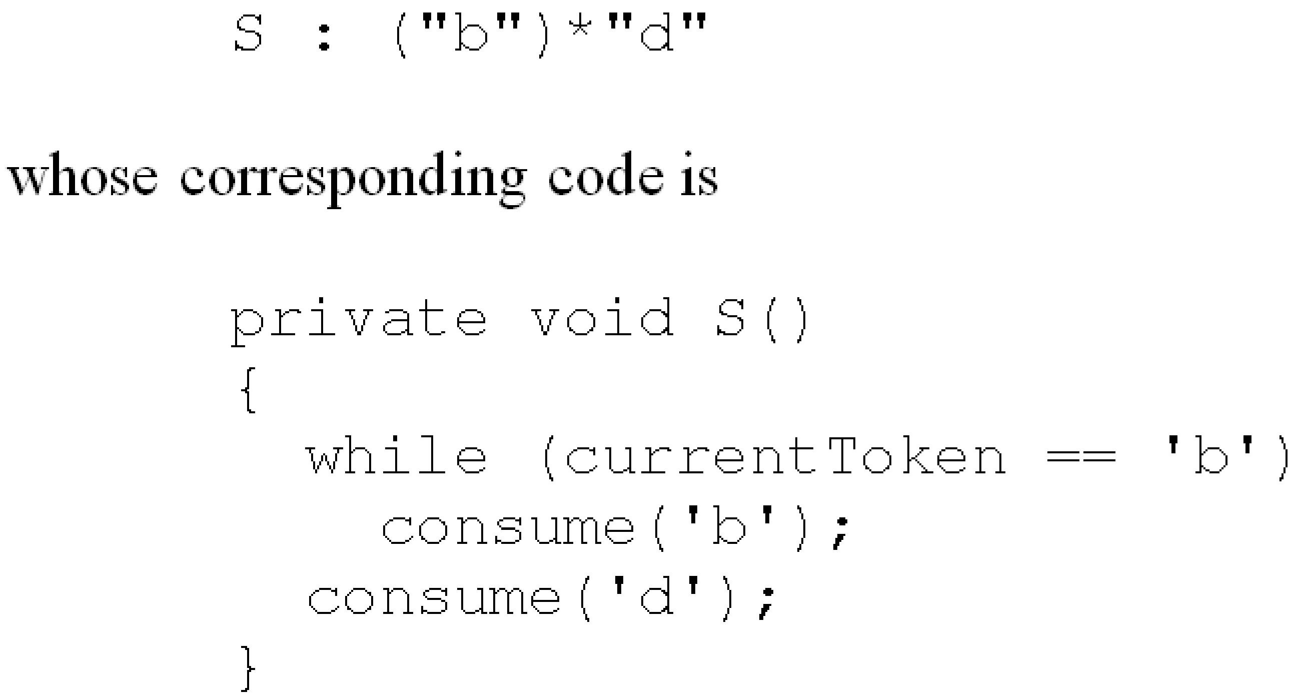
0

# Equivalent iterative approach



```
void nr1(int n)
{
  while (n > 0)
  {
    System.out.println(n);
    n = n - 1;
  }
  System.out.println("goodbye");
}
```

# Translating the star operator

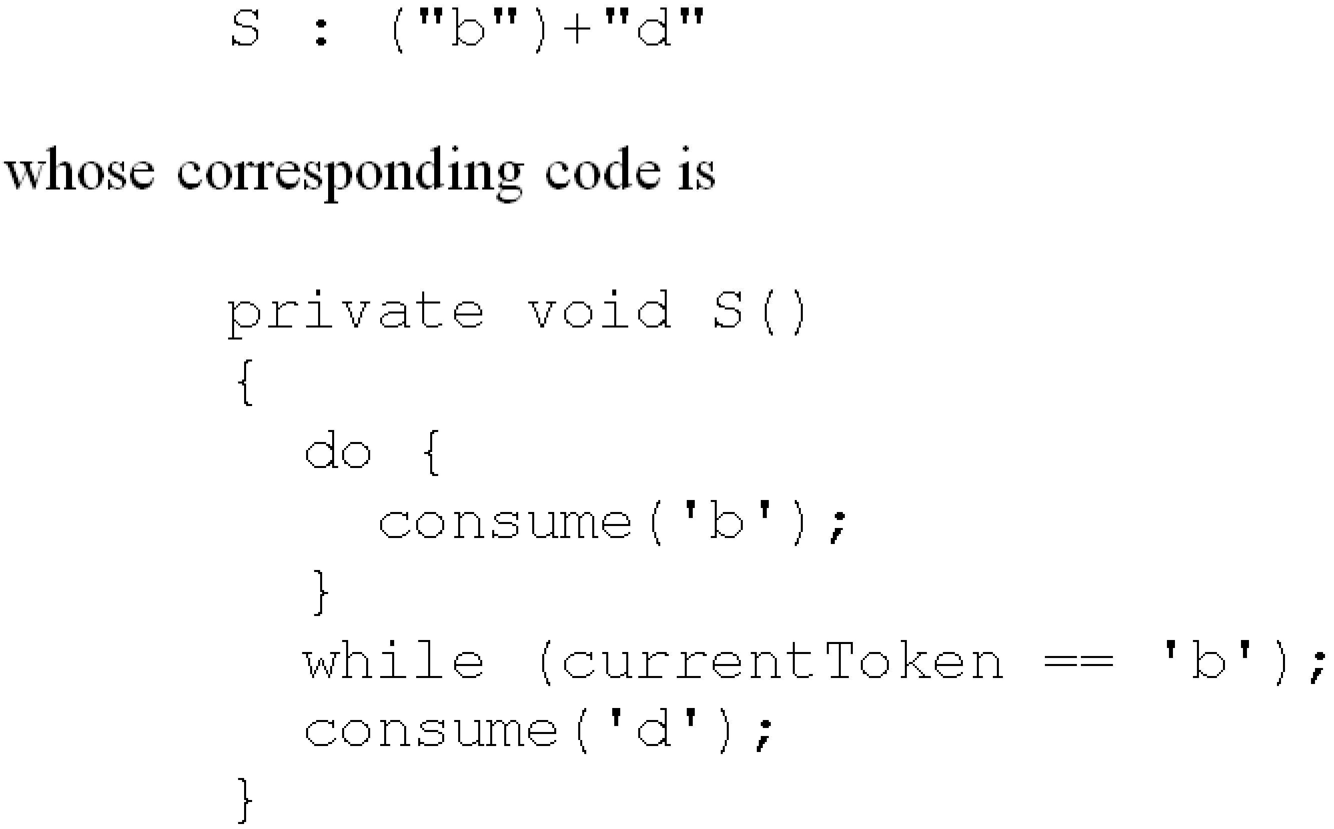


```
S : ("b")*"d"
```

whose corresponding code is

```
private void S()
{
  while (currentToken == 'b')
    consume('b');
  consume('d');
}
```

# Translating the plus operator

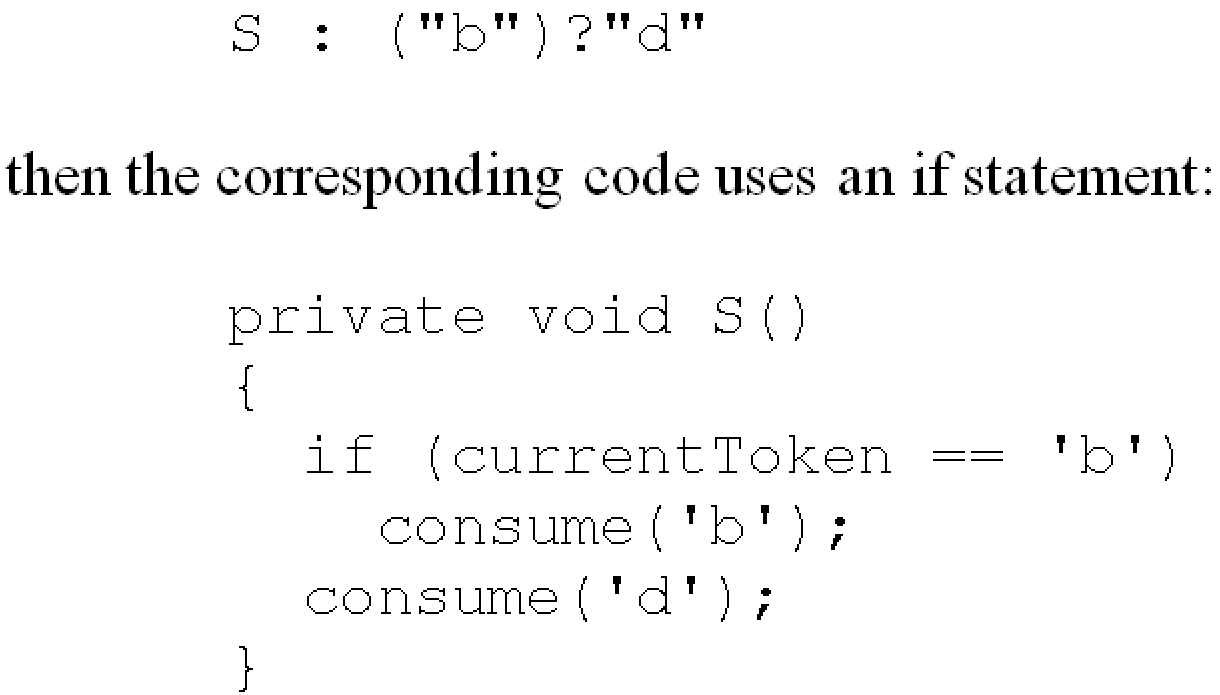


```
S : ("b")+"d"
```

whose corresponding code is

```
private void S()
{
  do {
    consume('b');
  }
  while (currentToken == 'b');
  consume('d');
}
```

# Translating the question mark operator

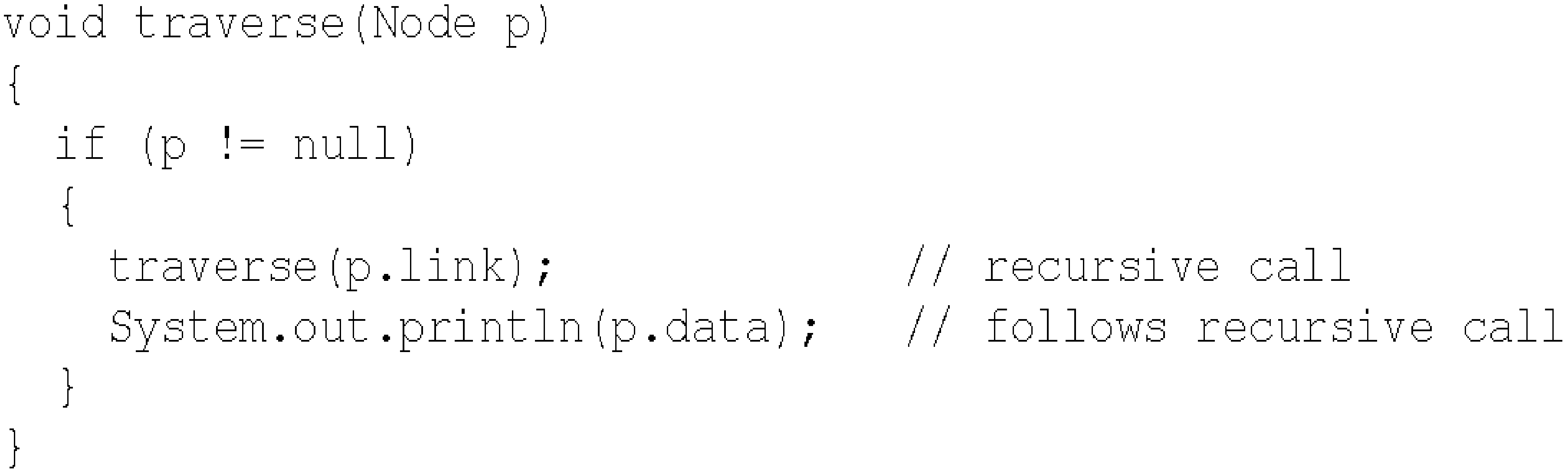


```
S : ("b")?"d"
```

then the corresponding code uses an if statement:

```
private void S()
{
  if (currentToken == 'b')
    consume('b');
  consume('d');
}
```

# Doing things backwards



```
void traverse(Node p)
{
  if (p != null)
  {
    traverse(p.link);              // recursive call
    System.out.println(p.data);    // follows recursive call
  }
}
```