Started on Thursday, 17 August 2023, 14:06
State Finished
Completed on Thursday, 17 August 2023, 14:30
Time taken 23 mins 47 secs
Grade 8.0 out of 15.0 (**56.7%**)

Quiz navigation

Code Section
1 2 3

Theory Section
4 5 6 7

Show one page at a time
Finish review

Question 1
Incorrect
Mark 0.0 out of 3.0
Flag question

Suppose we wanted to write a scanner that can skip comments and match the specification of numbers, as expressed in Cocol as follows:

```
CHARACTERS
  digit    = "0123456789".
TOKENS
  number = digit { digit } .   /* digits only */

COMMENTS FROM "/*" TO "*/"
IGNORE CHR(1) .. CHR(32)   /* character handler returns CHR(0) as EOF */
```

To enable comments to be skipped, what code would you write to replace the comment "**// deal with comments**" in the skeleton file below?

```
static void getSym() {      // Scans for next sym
  StringBuilder symLex = new StringBuilder();
  int symKind = noSym;        // assume the worst!
  while (ch != EOF && ch == ' ') getChar();  // ignore whitespace
    // deal with comments

  else {
    if (Character.IsDigit(ch))
    // deal with numbers

    else if (ch == EOF) symKind = EOFSym;
    sym = new Token(symKind, symLex.toString());
  }
```

Select one:

a.
```
if (ch == '(') {          // comment
  do getChar(); while (ch != '*' && ch != EOF);
  if (ch == '*') {
    getSym(); getChar(); return; }
  else output.write("unclosed comment");
}
```

b.
```
if (ch == '(') {          // comment
  do getChar(); while (ch != '*' && ch != EOF);
  if (ch == '*') {
    getChar(); getSym(); return; }
  else output.write("unclosed comment");
}
```

c.
```
if (ch == '(') {          // comment
  while (ch != '*' && ch != EOF) getChar();
  if (ch == '*') {
    getChar(); return; }
  else output.Write("unclosed comment");
  }
```

d.
```
if (ch == '(') {          // comment
  getChar();
  if (ch == '*') {
    getChar();
    char lastCh;
    do {
      lastCh = ch;
      getChar();
    } while (!(lastCh == '*' && ch == ')') && ch != EOF);
    if (ch == EOF)
      ReportError("unclosed comment");  // next sym would be EOFSym
    getChar(); getSym(); return; }
```

Your answer is incorrect.
The correct answer is: `if (ch == '(') { // comment
  do getChar(); while (ch != '*' && ch != EOF);
  if (ch == '*') {
    getChar(); getSym(); return; }
  else output.write("unclosed comment");
  }`

Question 2
Correct
Mark 3.0 out of 3.0
Flag question

Suppose we have a grammar which included the following productions:

```
PRODUCTIONS
  A = [ B ] '(' number { '+' number [ B ] ')' .
  B = '-' number ( '(' | '.' ) .
```

Assuming that you have accept() and abort() routines like those you used last week, and a scanner getSym() that can recognise tokens that might be described by the following constant identifiers:

EOFSym, noSym, minusSym, numSym, lparenSym, rparenSym, dotSym;

which of the following code snippets shows the BEST way of implementing a recursive descent parsing method for production A?

Select one:

a.
```
static void A () {
  // A = [ B ] '(' number { '+' number [ B ] ')' .

  {
    if (sym.kind != lparenSym )
      B();
    accept(lparenSym, "(" expected");
    accept(numSym, "number expected");
    while (sym.kind == lparenSym)
    { if (sym.kind == plusSym) {
        getSym(); accept(numSym, "number expected");
      }
      else B();
    }
    accept(rparenSym, ")" expected);
  }
```

b.
```
static void A () {
  // A = [ B ] '(' number { '+' number [ B ] ')' .

  {
    if (sym.kind != lparenSym)
      B();
    accept(lparenSym, "(" expected");
    accept(numSym, "number expected");
    while (sym.kind == plusSym || sym.kind == minusSym)
    { if (sym.kind == plusSym) {
        accept(numSym, "number expected");
      }
      else B();
    }
    accept(rparenSym, ")" expected);
  }
```

c.
```
static void A () {
  // A = [ B ] '(' number { '+' number [ B ] ')' .

  {
    if (sym.kind == minusSym || sym.kind == dotSym )
      B();
    accept(lparenSym, "(" expected");
    accept(numSym, "number expected");
    while (sym.kind == plusSym || sym.kind == minusSym
           || sym.kind == dotSym)
    { if (sym.kind == plusSym) {
        getSym(); accept(numSym, "number expected");
      }
      else B();
    }
    accept(rparenSym, ")" expected);
  }
```

d.
```
static void A () {
  // A = [ B ] '(' number { '+' number [ B ] ')' .

  {
    if (sym.kind == minusSym || sym.kind == dotSym )
      B();
    accept(lparenSym, "(" expected");
    accept(numSym, "number expected");
    while (sym.kind == plusSym || sym.kind == minusSym
           || sym.kind == dotSym)
    { if (sym.kind == plusSym) {
        accept(numSym, "number expected");
      }
      else B();
    }
    accept(rparenSym, ")" expected);
  }
```

e. None of the answers given is correct

Your answer is correct.
The correct answer is: `static void A () {
  // A = [B] '(' number { '+' number [B] ')' .

  {
    if (sym.kind == minusSym || sym.kind == dotSym)
      B();
    accept(lparenSym, "(" expected");
    accept(numSym, "number expected");
    while (sym.kind == plusSym || sym.kind == minusSym
           || sym.kind == dotSym)
    { if (sym.kind == plusSym) {
        getSym(); accept(numSym, "number expected");
      }
      else B();
    }
    accept(rparenSym, ")" expected);
  }`

Question 3
Incorrect
Mark 0.0 out of 2.0
Flag question

Suppose we have a grammar which included the following productions (where number is described by a Cocol token):

```
PRODUCTIONS
  A = [ B ] '(' number { '+' number [ B ] ')' .
    B = '-' number ( '(' | '.' ) .
```

Assuming that you have accept() and abort() routines like those you used last week, and a scanner getSym() that can recognise tokens that might be described by the constant identifiers:

EOFSym, noSym, plusSym, minusSym, numSym, lparenSym, rparenSym, dotSym

which of the following code snippets is the most correct version of a recursive descent parsing method to implement production B?

Select one:

a.
```
static void B () {
  // B = '-' number ( '(' | '.' .
  if (sym.kind == minusSym) {
    getSym();
    accept(numSym, "number expected");
    accept(lparenSym, " expected";
    else getSym();
  }
```

b.
```
static void B () {
  // B = '-' number ( '(' | '.' .
  if (sym.kind == minusSym) {
    getSym();
    accept(numSym, "number expected"); getSym();
    accept(lparenSym, " expected"); getSym();
  } else accept(lparenSym, " expected"); getSym();
}
```

c.
```
static void B () {
  // B = '-' number ( '(' | '.' .
    switch (sym.kind)
    {   case minusSym:   getSym();
                accept(numSym, "number expected");
                accept(lparenSym, " expected");
                break;
        case periodSym:  accept(periodSym, " expected");
                break;
        default:    abort("Incorrect start to B");  break;
    }
  }
```

d.
```
static void B () {
  // B = '-' number ( '(' | '.' .
    switch (sym.kind)
    {   case minusSym:   getSym();
                accept(numSym, "number expected");
                accept(periodSym, " expected");
                break;
        case periodSym:  accept(periodSym, " expected");
                break;
        default:    abort("Incorrect start to B");
                break;
    } }
```

e.
```
static void B () {
  // B = '-' number ( '(' | '.' .
    switch (sym.kind)
    {   case minusSym:   getSym();
                number();
                accept(periodSym, " expected");
                break;
        case periodSym:  accept(periodSym, " expected");
                break;
        default:    abort("Incorrect start to B");
                break;
    } }
```

Your answer is incorrect.
The correct answer is: `static void B () {
  // B = '-' number ('(' | '.' .
    switch (sym.kind)
    {   case minusSym: getSym();
                accept(numSym, "number expected");
                accept(periodSym, " expected");
                break;
        case periodSym: accept(periodSym, " expected");
                break;
        default: abort("Incorrect start to B");
                break;
    } }`

Question 4
Correct
Mark 1.0 out of 1.0
Flag question

True or False: An ambiguous grammar can always be parsed with an LL(2) parser.

Select one:

- True
- False ✔

The correct answer is 'False'.

Question 5
Correct
Mark 1.0 out of 1.0
Flag question

True or False: An LL(1) grammar can never be ambiguous.

Select one:

- True ✔
- False

The correct answer is 'True'.

Question 6
Complete
Mark 0.5 out of 2.0
Flag question

In your own words, explain what the purpose is of using the SYNC keyword in Cocol.

The SYNC keyword allows for Cocol to detect an error and then move past it to parse the rest of the grammar instead of just aborting at the first error found.

Comment:

Question 7
Correct
Mark 3.0 out of 3.0
Flag question

Suppose we wanted to write a scanner that can parse the specification of numbers, as expressed in Cocol as follows:

```
CHARACTERS
  digit    = "0123456789".
IGNORE CHR(1) .. CHR(32)   /* character handler returns CHR(0) as EOF */
TOKENS
  number = digit { digit } .   /* floating point number */
```

To recognise numbers, what code would you write to replace the comment "**// deal with numbers**" in the skeleton file below?

```
static void getSym() {      // Scans for next sym
  StringBuilder symLex = new StringBuilder();
  int symKind = noSym;        // assume the worst!

  // code here to ignore whitespaces and/or comments

  if (Character.isDigit(ch)
    {  // deal with numbers
    }

  else if (ch == EOF) symKind = EOFSym;
  sym = new Token(symKind, symLex.toString());
}
```

Select one:

a.
```
do {   symLex.append(ch);   getChar();
    while (Character.isDigit(ch));
if (ch == '.') { symLex.append(ch);  getChar(); }
while (Character.isDigit(ch))
    { symLex.append(ch);  getChar();  }
symKind = numSym;
```

b.
```
do {   symLex.append(ch);   getChar();
    while (Character.isDigit(ch));
if  (ch == '.')
{   symLex.append(ch);  getChar();
    if (Character.isDigit(ch))
    { do {   symLex.append(ch); getChar();
           while (Character.isDigit(ch));
      }
      else symKind = noSym;
    }
    else symKind = noSym;
```

c.
```
do {   symLex.append(ch);   getChar();
    while (Character.isDigit(ch));
if (ch == '.')
{   symLex.append(ch);  getChar();
    while  (Character.isDigit(ch))
    { symLex.append(ch);  getChar();  }
    symKind = noSym;
}
else symKind = numSym;
```

d.
```
do {   symLex.append(ch);   getChar();
    while (Character.isDigit(ch));
if (ch == '.')
{   symLex.append(ch);  getChar();
    while  (Character.isDigit(ch))
    { symLex.append(ch);  getChar();  }
    symKind = noSym;
}
else symKind = numSym;
```

Your answer is correct.
The correct answer is: `do {   symLex.append(ch);   getChar();
    while (Character.isDigit(ch));
if (ch == '.')
{   symLex.append(ch); getChar();
    while (Character.isDigit(ch))
    { symLex.append(ch); getChar(); }
    symKind = numSym;
}
else symKind = noSym;`

Finish review