Computer Science 3 - 2011

Programming Language Translation

Practical for Week 23, beginning 3 October 2011 - Solutions

This practical was done fairly well by all but a few groups, These parsers are not hard to write -but, alas, they are also easy to get wrong (putting `getSym()` calls in the wrong places!). One point that I noticed was that some people were driving their parsers from the back, so to speak. Given a construction like

```
A = { "start" Something } "follow" .
```

it is far better to produce a parser routine like

```
while (sym.kind == startSym) { getSym(); Something(); } Accept(followSym);
```

than one like

```
while (sym.kind != followSym) { getSym(); Something(); } Accept(followSym);
```

for the simple reason that there might be something wrong with `Something`.

Complete source code for solutions to the prac are available on the WWW pages in the file `PRAC23A.ZIP` or `PRAC23AC.ZIP` (C# version).

The scanner was not well done by some people. Comments are trickier than they look, so if you did not get that section right study my solution below carefully and see how they could be handled. You must be able to handle comments that are never closed, comments that follow one another with no tokens in between them, and comments that look as they are about to finish, but then do not (like (* comment with wrong parenthesis *)). Did your scanner handle comments like {} or {}{} or {{ } } or {a}{b} or mistakes like {)?

Secondly, the scanning of a `Name` is trickier than it looks - if a name contains ' or - characters one needs to check that the next character is an uppercase letter. What if it isn't? Do we continue to scan and include a lower-case letter or another ' or - that follws the first one or do we abort scanning, leaving the first incorrect character to be picked up as the first character of the next toke to be scanned. For example, what should a scanner do when faced with input like

```
Name                 { correct }
FirstName LastName   { correct - pairs of two names in succession with no separator (dangerous) }
O'Malley             { correct }
Mary-Beth            { correct }

Name-you             { incorrect, but how is it scanned? }
Name--You            { incorrect, but how is it scanned? }
Name-'You            { incorrect, but how is it scanned? }
Name-'you            { incorrect, but how is it scanned? }
Name'-'you           { incorrect, but how is it scanned? }
```

You might like to ponder this one and then look at the various alternatives given in the scanner that follws and try to decide which is the best one. In any event, the scanner should be able to return a `noSym` if an incorrect name is detectd. (I only thought of all this after examining the various submissions, some of which intrigued me, so well done. It's fun to be made to think by what my students suggest!)

```
// ++++++++++++++++++++++++++++++ Scanner +++++++++++++++++++++++++++++++++

// Declaring sym as a global variable is done for expediency - global variables
// are not always a good thing

static Token sym;

static int literalKind(StringBuilder lex, int defaultSym) {
  String s = lex.toString();
  if (s.equals("Children"))       return childrenSym;
  if (s.equals("Grandchildren"))  return grandchildrenSym;
  if (s.equals("Grandparents"))   return grandparentsSym;
  if (s.equals("Parents"))        return parentsSym;
  if (s.equals("Surname"))        return surnameSym;
  if (s.equals("brother"))        return brotherSym;
```

```
      if (s.equals("daughter"))       return daughterSym;
      if (s.equals("deceased"))       return deceasedSym;
      if (s.equals("father"))         return fatherSym;
      if (s.equals("husband"))        return husbandSym;
      if (s.equals("mistress"))       return mistressSym;
      if (s.equals("mother"))         return motherSym;
      if (s.equals("of"))             return ofSym;
      if (s.equals("partner"))        return partnerSym;
      if (s.equals("sister"))         return sisterSym;
      if (s.equals("son"))            return sonSym;
      if (s.equals("wife"))           return wifeSym;
      return defaultSym;
    }

    static void getSym() {
    // Scans for next sym from input
      while (ch > EOF && ch <= ' ') getChar();

      if (ch == '{') {  // must be a comment
        int level = 1;
        do {
          getChar();
          if (ch == '}') level--; else if (ch == '{') level++;
        } while (level > 0 && ch != EOF);
        if (ch == EOF)
          reportError("unclosed comment"); // sym will be EOFSym
        getChar(); getSym(); return;
      }

  /*  ================ alternatively

      if (ch == '{') {  // must be a comment
        int level = 1;
        getChar();
        do {
          if (ch == '}') level--; else if (ch == '{') level++;
          getChar();
        } while (level > 0 && ch != EOF);
        if (ch == EOF)
          reportError("unclosed comment"); // sym will be EOFSym
        getSym(); return;
      }

  ==================== */

      else {
        StringBuilder symLex = new StringBuilder();
        int symKind = noSym;
        if (Character.isUpperCase(ch)) { // looks like a name will be scanned
          symKind = nameSym;
          symLex.append(ch); getChar();

  // one way of proceeding:

          do {
            if (ch == '-' || ch == '\'' ) {
              symLex.append(ch); getChar();
              if (!Character.isUpperCase(ch)) symKind = noSym;
            }
            symLex.append(ch); getChar();
          } while (symKind == nameSym && (ch == '-' || ch == '\'' || Character.isLowerCase(ch)));
          if (symKind != noSym) symKind = literalKind(symLex, nameSym);
        }

  // */

  /* another way of proceeding

          do {
            if (ch == '-' || ch == '\'' ) {
              symLex.append(ch); getChar();
              if (!Character.isUpperCase(ch)) symKind = noSym;
            }
            symLex.append(ch); getChar();
          } while (ch == '-' || ch == '\'' || Character.isLowerCase(ch));
          if (symKind != noSym) symKind = literalKind(symLex, nameSym);
        }
  */

  /* yet another way of proceeding
```

```
          while (symKind == nameSym && (ch == '-' || ch == '\'' || Character.isLowerCase(ch))) {
            if (Character.isLowerCase(ch)) {
              symLex.append(ch); getChar();
            }
            else if (ch == '-' || ch == '\'' ) {
              symLex.append(ch); getChar();
              if (Character.isUpperCase(ch)) {
                symLex.append(ch); getChar();
              }
              else symKind = noSym;
            }
          } // while
          if (symKind != noSym) symKind = literalKind(symLex, nameSym);
        }
  */

        else if (Character.isLowerCase(ch)) { // presumably a word
          do {
            symLex.append(ch); getChar();
          } while (Character.isLetter(ch));
          symKind = literalKind(symLex, wordSym);
        }

        else if (Character.isDigit(ch)) {     // presumably a number
          do {
            symLex.append(ch); getChar();
          } while (Character.isDigit(ch));
          symKind = numSym;
        }

        else {                                // presumably a single character token
          symLex.append(ch);
          switch (ch) {
            case EOF:
              symLex = new StringBuilder("EOF");      // special representation
              symKind = EOFSym; break;                // no need to getChar here, of course
            case ',':
              symKind = commaSym; getChar(); break;
            case ':':
              symKind = colonSym; getChar(); break;
            case '(':
              symKind = lparenSym; getChar(); break;
            case ')':
              symKind = rparenSym; getChar(); break;
            case '[':
              symKind = lbrackSym; getChar(); break;
            case ']':
              symKind = rbrackSym; getChar(); break;
            case '=':
              symKind = equalSym; getChar(); break;
            case '.':
              symKind = periodSym; getChar(); break;
            default :                                 // even here, scan to next character
              symKind = noSym; getChar(); break;
          }
        }
        sym = new Token(symKind, symLex.toString());
      }
    } // getSym
```

Here is a simple "sudden death" parser, devoid of error recovery:

```
    // +++++++++++++++++++++++++++++++ Parser +++++++++++++++++++++++++++++++++++

    static IntSet FirstRelative   = new IntSet(brotherSym, daughterSym, fatherSym, husbandSym, mistressSym,
                                               motherSym, partnerSym, sisterSym, sonSym, wifeSym);
    static IntSet FirstGeneration = new IntSet(childrenSym, grandchildrenSym, grandparentsSym, parentsSym);
    static IntSet FirstPossession = new IntSet(numSym, wordSym);

    static void accept(int wantedSym, String errorMessage) {
    // Checks that lookahead token is wantedSym
      if (sym.kind == wantedSym) getSym(); else abort(errorMessage);
    } // accept

    static void accept(IntSet allowedSet, String errorMessage) {
    // Checks that lookahead token is in allowedSet
      if (allowedSet.contains(sym.kind)) getSym(); else abort(errorMessage);
    } // accept
```

```
static void Family() {
// Family = { Generation } Surname { Generation } { Possession }  "." .
  while (FirstGeneration.contains(sym.kind)) Generation();
  Surname();
  while (FirstGeneration.contains(sym.kind)) Generation();
  while (FirstPossession.contains(sym.kind)) Possession();
} // Family

static void Surname() {
// Surname   = "Surname" ":" name { name } .
  accept(surnameSym, " Surname expected");
  accept(colonSym, " : expected");
  accept(nameSym, " name expected");
  while (sym.kind == nameSym) getSym();
} // Surname

static void Generation() {
// Generation = ( "Parents" | "Grandparents" | "Children" | "Grandchildren" ) ":" NameList .
  accept(FirstGeneration, "Parents, Grandparents, Children or Grandchildren expected");
  accept(colonSym, " : expected");
  NameList();
} // Generation

static void NameList() {
// NameList = OnePerson { "," OnePerson } .
  OnePerson();
  while (sym.kind == commaSym) {
    getSym();
    OnePerson();
  }
} // NameList

static void OnePerson() {
// OnePerson = name { name } [ "(" "deceased" ")" ] { Description } [ Spouse ] .
  accept(nameSym, " name expected");
  while (sym.kind == nameSym) getSym();
  if (sym.kind == lparenSym) {
    getSym();
    accept(deceasedSym, " deceased expected");
    accept(rparenSym, " } expected");
  }
  while (sym.kind == lbrackSym) Description();
  if (sym.kind == equalSym) Spouse();
} // OnePerson

static void Spouse() {
// Spouse = "=" name { name } .
  accept(equalSym, " = expected");
  accept(nameSym, " name expected");
  while (sym.kind == nameSym) getSym();
} // Spouse

static void Description() {
// Description = "[" Relative "of" OnePerson "]" .
  accept(lbrackSym, " [ expected");
  Relative();
  accept(ofSym, " of expected");
  OnePerson();
  accept(rbrackSym, " ] expected");
} // Description

static void Relative() {
// Relative =   "son"   | "daughter" | "mother"   | "father" | "sister"
//            | "brother" | "wife" | "husband"   | "partner" | "mistress" .
  accept(FirstRelative, " unknown relationship");
} // Relative

static void Possession() {
// Possession = [ number ] word { word } .
  if (sym.kind == numSym) getSym();
  accept (wordSym, " word expected");
  while (sym.kind == wordSym)  getSym();
} // Possession
```

A very comon mistake is to overlook and condone (not detect) some errors. Compare `Constant` above with this (dangerous) version. If you can't see why I don't particularly like the code below, you had better come to ask!

```
static void Surname() {
// Surname   = "Surname" ":" name { name } .
  getSym();
  accept(colonSym, " : expected");
  accept(nameSym, " name expected");
  while (sym.kind == nameSym) getSym();
} // Surname
```

The point to make is that a parsing method should not assume that it will always be called if its "precondition" is met. That *should* be the case, but remember that anyone can write a compiler if the user will never make mistakes - but users invariably *do* make mistakes. So if you have a production like

```
Something = "one" SomethingElse .
```

it is safer to code the parsing method as

```
void Something() }
  accept(oneSym, "one expected");
  SomethingElse();
}
```

and not as

```
void Something() {
  getSym();
  SomethingElse();
}
```

Of course, in this example, many of the methods *would* only have been called if the token had satisfied the precondition, as some sort of test would have been made in the caller. Another point that is easily missed can be illustrated by the production

```
Something = "one" FollowOne | "two" FollowTwo
```

If you code the parsing method as

```
void Something() {
  if (sym.kind == oneSym) {
    getSym(); FollowOne();
  }
  else {
    getSym(); FollowTwo();
  }
}
```

you run the risk of not detecting the error if `Something()` is called with `sym` corresponding to something other than "one" or "two". The code would be much better as

```
void Something() {
  if (sym.kind == oneSym) {
    getSym(); FollowOne();
  }
  else if (sym.kind == twoSym) {
    getSym(); FollowTwo();
  }
  else abort("invalid start to Something");
}
```

or as

```
void Something() {
  switch(sym.kind)
    case oneSym :
      getSym(); FollowOne(); break;
    case twoSym) :
      getSym(); FollowTwo(); break;
    default :
      abort("invalid start to Something"); break;
  }
}
```

although you could almost "get away" with

```
void Something() {
  if (sym.kind == oneSym) {
    getSym(); FollowOne();
  }
  else {
    accept(twoSym, "invalid start to Something"); FollowTwo();
  }
}
```

because an error message will be generated if the token is not one of "one" or "two".

If in doubt, use the `accept()` method rather than a simple `getSym()` - and make sure all your `switch` statements *always* have a `default` clause.

You might be interested in the following variation on the `Spouse()` and parsers, which illustrate the use of a do-while loop in a safe situation synchronisation.

```
static void Spouse() {
// Spouse = "=" name { name } .
  accept(equalSym, " = expected");
  do
    accept(nameSym, " name expected");
  while (sym.kind == nameSym);
} // Spouse

static void Possession() {
// Possession = [ number ] word { word } .
  if (sym.kind == numSym) getSym();
  do
    accept (wordSym, " word expected");
  while (sym.kind == wordSym);
} // Possession
```

Finally, notice that it was useful to overload the `accept()` method to have a version that tests a single token as well as a second version that tests for membership of a set. This was not mentioned in The Book, although the code was supplied in the kit.

```
static void accept(int wantedSym, String errorMessage) {
// Checks that lookahead token is wantedSym
  if (sym.kind == wantedSym) getSym(); else abort(errorMessage);
}

static void accept(IntSet allowedTokens, String errorMessage) {
// Checks that lookahead token is in a set of allowedTokens
  if (allowedTokens.contains(sym.kind)) getSym(); else abort(errorMessage);
}
```