Computer Science 3 - 2009

Programming Language Translation

Practical for Week 23, beginning 5 October 2009 - Solutions

This practical was done well by all but a few groups, 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 - including the error recovery additions that you were asked to think about - are available on the WWW pages in the file `PRAC23A.ZIP` or `PRAC23AC.ZIP` (C# version).

The scanner was reasonably done by most 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 should be handled. The trick is to be able to handle comments that are never closed as well as comments that follow one another with no tokens in between them.

```
static int literalKind(StringBuilder lex) {
  String s = lex.toString();
  if (s.equals("music"))        return musicSym;
  if (s.equals("advert"))       return advertSym;
  if (s.equals("zuma"))         return zumaSym;
  if (s.equals("malema"))       return malemaSym;
  if (s.equals("zille"))        return zilleSym;
  if (s.equals("corruption"))   return corruptSym;
  if (s.equals("semenya"))      return semenyaSym;
  if (s.equals("host"))         return hostSym;
  if (s.equals("listener"))     return listenerSym;
  if (s.equals("maxmin"))       return maxminSym;
  if (s.equals("temperatures")) return tempSym;
  return wordSym;
}

static void getSym() {
// Scans for next sym from input
  while (ch > EOF && ch <= ' ') getChar();
  if (ch == '{') {  // must be a comment
    getChar();
    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;
  }
  else {
    StringBuilder symLex = new StringBuilder();
    int symKind;
    if (Character.isLetter(ch)) {
      do {
        symLex.append(ch); getChar();
      } while (Character.isLetter(ch));
      symKind = literalKind(symLex);
    }
    else if (Character.isDigit(ch)) {
      do {
        symLex.append(ch); getChar();
      } while (Character.isDigit(ch));
      symKind = numSym;
    }
    else {
```

```
      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 = periodSym; getChar(); break;
        default :
          symKind = noSym; getChar(); break;
      }
    }
    sym = new Token(symKind, symLex.toString());
  }
}
```

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

```
static IntSet
  FirstRadio     = new IntSet(hostSym, musicSym, advertSym),
  FirstNewsItem  = new IntSet(malemaSym, zumaSym, zilleSym, corruptSym, wordSym),
  FirstRestStory = new IntSet(malemaSym, zumaSym, semenyaSym, wordSym),
  FirstFiller    = new IntSet(musicSym, advertSym);

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 acceptable, String errorMessage) {
// Checks that lookahead sym.kind is in acceptable set of tokens
  if (acceptable.contains(sym.kind)) getSym(); else abort(errorMessage);
}

static void Radio() {
// Radio = { TalkShow | "music" | "advert" [ NewsBulletin ] } EOF .
  while (FirstRadio.contains(sym.kind)) {
    switch (sym.kind) {
      case hostSym :
        TalkShow(); break;
      case musicSym :
        getSym(); break;
      case advertSym :
        getSym();
        if (FirstNewsItem.contains(sym.kind)) NewsBulletin();
        break;
    }
  }
  accept(EOFSym, "EOF expected");
}

static void NewsItem() {
// NewsItem = "zuma" [ "malema" ] | "malema" "zuma" | "zille" | "randFalls" | Story .
  switch (sym.kind) {
    case zumaSym :
      getSym(); if (sym.kind == malemaSym) getSym();
      break;
    case malemaSym :
      getSym(); accept(zumaSym, "Zuma expected");
      break;
    case zilleSym :
    case corruptSym :
      getSym();
      break;
    case wordSym :
      Story();
      break;
    default :
      abort("Unacceptable start to news item");
      break;
  }
}
```

```
static void NewsBulletin() {
// NewsBulletin = NewsItem { NewsItem } [ Weather ] Filler .
  NewsItem();
  while (FirstNewsItem.contains(sym.kind)) NewsItem();
  if (sym.kind == maxminSym) Weather();
  Filler();
}

static void Story() {
// Story = word { word | "semenya" | "zuma" | "malema" } "." .
  accept(wordSym, "word expected");
  while (FirstRestStory.contains(sym.kind)) getSym();
  accept(periodSym, ". expected");
}

static void TalkShow() {
// TalkShow = Host { Listener Host } .
  Host();
  while (sym.kind == listenerSym) {
    Listener(); Host();
  }
}

static void Host() {
// Host = "host" Opinion .
  accept(hostSym, "host expected");
  Opinion();
}

static void Listener() {
// Listener = "listener" Opinion .
  accept(listenerSym, "listener expected");
  Opinion();
}

static void Opinion() {
// Opinion = Story .
  Story();
}

static void Filler() {
// Filler = "music" | "advert"  .
  accept(FirstFiller, "unacceptable filler item");
}

static void Weather() {
// Weather = "maxmin" "temperatures" OneTemp { "," OneTemp }.
  accept(maxminSym, "max/min expected");
  accept(tempSym, "temperarures expected");
  OneTemp();
  while (sym.kind == commaSym) {
    getSym();
    OneTemp();
  }
}

static void OneTemp() {
// OneTemp = word number number .
  accept(wordSym, "area expected");
  accept(numSym, "min temperature expected");
  accept(numSym, "max temperature expected");
}
```

To incorporate error recovery one can go to the whole extent of introducing calls to a test routine at the start and end of every subparser. But one can probably get away with something a lot simpler:

```
static IntSet
  FirstRadio     = new IntSet(hostSym, musicSym, advertSym),
  FirstSync      = new IntSet(hostSym, musicSym, advertSym, EOFSym),
  FirstNewsItem  = new IntSet(malemaSym, zumaSym, zilleSym, corruptSym, wordSym),
  FollowNews     = new IntSet(malemaSym, zumaSym, zilleSym, corruptSym, wordSym, musicSym, advertSym, maxminSym),
  FirstRestStory = new IntSet(malemaSym, zumaSym, semenyaSym, wordSym),
  FirstFiller    = new IntSet(musicSym, advertSym);

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

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

static void test(IntSet allowed, IntSet beacons, String errorMessage) {
// Test whether current Sym is in Allowed, and recover if not
  if (allowed.contains(sym.kind)) return;
  reportError(errorMessage);
  IntSet stopSet = allowed.union(beacons);
  while (!stopSet.contains(sym.kind)) getSym();
  output.writeLine();   // for demonstration purposes only
  output.writeLine("-------------- sync ---------- ");
}

static void Radio() {
// Radio = { TalkShow | "music" | "advert" [ NewsBulletin ] } EOF .
  while (FirstRadio.contains(sym.kind)) {
    switch (sym.kind) {
      case hostSym :
        TalkShow(); break;
      case musicSym :
        getSym(); break;
      case advertSym :
        getSym();
        if (FirstNewsItem.contains(sym.kind)) NewsBulletin();
        break;
    }
    test(FirstSync, new IntSet(), "Invalid sequence of programmes");  // --------------------
  }
  accept(EOFSym, "EOF expected");
}

static void NewsItem() {
// NewsItem = "zuma" [ "malema" ] | "malema" "zuma" | "zille" | "randFalls" | Story .
  switch (sym.kind) {
    case zumaSym :
      getSym(); if (sym.kind == malemaSym) getSym();
      break;
    case malemaSym :
      getSym(); accept(zumaSym, "Zuma expected");
      break;
    case zilleSym :
    case corruptSym :
      getSym();
      break;
    case wordSym :
      Story();
      break;
    default :
      reportError("Unacceptable start to news item");                // --------------------
      break;
  }
  test(FollowNews, new IntSet(), "invalid item after NewsItem");      // --------------------
}
```

Task 6 asked you to consider why and whether, when the grammar was non-LL(1), the parser would still behave satisfactorily.

It is non-LL(1) because Rule 2 is broken in the productions

```
NewsBulletin = NewsItem { NewsItem } [ Weather ] Filler .
NewsItem     = "zuma" [ "malema" ] | "malema" "zuma" | "zille" | "corruption" | Story .
```

The component `[ "malema" ]` is nullable, and its *FIRST* set contains `malema` as does its *FOLLOW* set, which includes all the elements of *FIRST(NewsItem)* = { `zuma, malema, zille, corruption, word`}.

The grammar is ambiguous too. Because of the way in which the LL(1) rules are broken in this case, in theory one can parse a *NewsBulletin* reading `zuma malema zuma music` as either `(zuma malema) (zuma) (music)` or as `(zuma) (malema zuma) (music)`. The parser described here will, however, choose the first of these and bind `malema` to the first zuma, and will not bind the last `zuma` to `malema`. It's really another "dangling else" situation. Whereas in the real dangling else situation it makes good sense to bind an *else* to the most recent *if*, the same might not apply here - some stories involving `malema` will doubtless be part of the story introduced by `zuma`, while others might be ones that start off with `malema` and then `only` drag `zuma` in later.