Computer Science 3 - 2008

Programming Language Translation

Practical for Week 23, beginning 29 September 2008 - Solutions

There were some very good (but no perfect) solutions handed in, but several people had not absorbed all the detail. This showed up very clearly in the little prac test, which revealed, rather sadly and surprisingly, that there seemed to be some students who did not have the first clue about how to write a simple recursive descent parser for a tiny system of productions. Please study the solutions to the prac test carefully - they are all on the WWW pages for the course under the heading of "Prac Tests".

Another point that I noticed was that many people, both in the test and in the prac, 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 three possible solutions to the prac are available on the WWW pages in files `PRAC23A.ZIP` (Java) or `PRAC23AC.ZIP` (C#).

Here is the important part of the code for the scanner. Most people got a lot of this correct - except perhaps for the comment handling (which you had been told to read up about in the textbook, so that failing to get that right was just silly). There was some funny clumsy code dealing with the "escaped" tokens - please have a good look at what I suggest, and in particular that one had to "match" the quotation marks.

```
// +++++++++++++++++++++++  token kinds enumeration ++++++++++++++++++++++++

static final int
  noSym        =  0,
  EOFSym       =  1,
  SemicolonSym =  2,
  BarSym       =  3,
  LParenSym    =  4,
  LBrackSym    =  5,
  RParenSym    =  6,
  RBrackSym    =  7,
  ConcatSym    =  8,
  OptionSym    =  9,
  RepeatSym    = 10,
  PlusSym      = 11,
  HyphenSym    = 12,
  AtomSym      = 13,
  EscapedSym   = 14;

// ++++++++++++++++++++++++++++++ Scanner +++++++++++++++++++++++++++++++++

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

static Token sym;

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

// comment handlers have to be carefully written - we must check for unclosed
// comments, as well as making sure that we really do find a token.  Note the
// recursive call to getSym

  if (ch == '\\') {                            // skip comment
    do getChar(); while (ch != '\\' && ch != EOF);
```

```
      if (ch != EOF) {                            // try again
        getChar(); getSym(); return;
      }
      else {                                      // give up
        sym = new Token(EOFSym, "EOF");
        //  perhaps   abort("comment was not terminated");
        return;
      }
    }
    StringBuilder symLex = new StringBuilder();
    int symKind = noSym;
    symLex.append(ch);
    switch (ch) {
      case EOF  : symKind = EOFSym;
                  symLex = new StringBuilder("EOF"); break;  // no need to getChar here, of course
      case ';'  : symKind = SemicolonSym; getChar(); break;
      case '|'  : symKind = BarSym; getChar(); break;
      case '('  : symKind = LParenSym; getChar(); break;
      case '['  : symKind = LBrackSym; getChar(); break;
      case ')'  : symKind = RParenSym; getChar(); break;
      case ']'  : symKind = RBrackSym; getChar(); break;
      case '.'  : symKind = ConcatSym; getChar(); break;
      case '?'  : symKind = OptionSym; getChar(); break;
      case '*'  : symKind = RepeatSym; getChar(); break;
      case '+'  : symKind = PlusSym; getChar(); break;
      case '-'  : symKind = HyphenSym; getChar(); break;

  // escaped characters need special treatment  - we must check for matching
  // quote marks

      case '"'  :
      case '\'' : char quote = ch;
                  getChar(); symLex.append(ch);
                  if (ch < ' ') break;                  // escaped exclude control chars
                  getChar(); symLex.append(ch);
                  if (ch == quote) {
                    symKind = EscapedSym;               // else noSym already
                    GetChar();
                  }
                  break;

  // unusually, perhaps, the default option mops up all other possibilities as
  // valid.  Very often default handlers must mop up errors

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

and here is code for a simple parser with brutal error handling:

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

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

  static IntSet
    FirstExpression = new IntSet(AtomSym, EscapedSym, LBrackSym, LParenSym),
    FirstFactor     = new IntSet(AtomSym, EscapedSym, LBrackSym, LParenSym),
    FirstAtom       = new IntSet(AtomSym, EscapedSym),
    FirstOneRange   = new IntSet(AtomSym, EscapedSym),
    Iterators       = new IntSet(RepeatSym, OptionSym, PlusSym);

  static void RE () {
  // RE == { Expression ";" } EOF .
  // Note that we drive the loop on a FirstExpression check, and not on
  //    while (sym != EOFSym) ...
  // Parser routines should normally be driven positively on FIRST sets, not negatively
  // on the use of FOLLOW sets
    while (FirstExpression.contains(sym.kind)) {
      Expression(); accept(SemicolonSym, "; expected");
    }
    accept(EOFSym, "EOF expected");
  }

  static void Expression () {
  // Expression = Term { "|" Term } .
    Term();
```

```
  while (sym.kind == BarSym) {
    getSym(); Term();
  }
}

static void Term () {
// Term = Factor { [ "." ] Factor } .
  Factor();
  while (FirstFactor.contains(sym.kind) || sym.kind == ConcatSym) {
    if (sym.kind == ConcatSym) getSym();
    Factor();
  }
}

static void Factor () {
// Factor == Element [ "*" | "?" | "+" ] .
  Element();
  if (Iterators.contains(sym.kind)) getSym();
}

static void Element () {
// Element == Atom | Range | "(" Expression ")" .
  if (FirstAtom.contains(sym.kind)) Atom();
  else
    if (sym.kind == LBrackSym) Range();
    else
      if (sym.kind == LParenSym) {
        getSym(); Expression(); accept(RParenSym, ") expected");
      }
      else abort("Invalid Start to Element ");
}

static void Range () {
// Range == "[" OneRange { OneRange } "]" .
  accept(LBrackSym, "[ expected");
  OneRange();
  while (FirstOneRange.contains(sym.kind)) OneRange();
  accept(RBrackSym, "] expected");
}

static void OneRange () {
// OneRange == Atom [ "-" Atom ] .
  Atom();
  if (sym.kind == HyphenSym) { getSym(); Atom(); }
}

static void Atom () {
// Atom = atomic | escaped .
// Note the use of a default option to mop up errors
  switch (sym.kind) {
    case AtomSym :     getSym(); break;
    case EscapedSym :  getSym(); break;
    default :          abort("Invalid Atom"); break;
  }
}
```

To incorporate "follow set" error recovery we could try something like this:

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

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

static void test(IntSet allowed, IntSet beacons, String errorMessage) {
  if (allowed.contains(sym.kind)) return;
  reportError(errorMessage);
  IntSet stopSet = allowed.union(beacons);
  while (!stopSet.contains(sym.kind)) getSym();
}

static IntSet
  FirstExpression = new IntSet(AtomSym, EscapedSym, LBrackSym, LParenSym),
  FirstElement    = new IntSet(AtomSym, EscapedSym, LBrackSym, LParenSym),
  FirstFactor     = new IntSet(AtomSym, EscapedSym, LBrackSym, LParenSym),
  FirstRestFactor = new IntSet(AtomSym, EscapedSym, LBrackSym, LParenSym, ConcatSym),
  FirstAtom       = new IntSet(AtomSym, EscapedSym),
  FirstOneRange   = new IntSet(AtomSym, EscapedSym),
  FirstRange      = new IntSet(LBrackSym),
```

```
    Iterators       = new IntSet(RepeatSym, OptionSym, PlusSym);

  static void RE (IntSet followers) {
  // RE == { Expression ";" } EOF .
    test(followers.union(FirstExpression), new IntSet(), "bad start to file"); // concession
    while (FirstExpression.contains(sym.kind)) {
      Expression(followers.union(new IntSet(SemicolonSym)));
      accept(SemicolonSym, "; expected");
    }
    accept(EOFSym, "EOF expected");
  }

  static void Expression (IntSet followers) {
  // Expression = Term { "|" Term } .
    Term(followers.union(new IntSet(BarSym)));
    while (sym.kind == BarSym) {
      getSym();
      Term(followers.union(new IntSet(BarSym)));
    }
  }

  static void Term (IntSet followers) {
  // Term = Factor { [ "." ] Factor } .
    Factor(followers.union(FirstRestFactor));
    while (FirstFactor.contains(sym.kind) || sym.kind == ConcatSym) {
      if (sym.kind == ConcatSym) getSym();
      Factor(followers.union(FirstRestFactor));
    }
  }

  static void Factor (IntSet followers) {
  // Factor == Element [ "*" | "?" | "+" ] .
    Element(followers.union(Iterators));
    if (Iterators.contains(sym.kind)) getSym();
  }

  static void Element (IntSet followers) {
  // Element == Atom | Range | "(" Expression ")" .
    test(FirstElement, followers, "Invalid Start to Element");
    if (FirstAtom.contains(sym.kind)) Atom(followers);
    else
      if (sym.kind == LBrackSym) Range(followers);
      else
        if (sym.kind == LParenSym) {
          getSym(); Expression(followers.union(new IntSet(RParenSym)));
          accept(RParenSym, ") expected");
        }
    test(followers, new IntSet(), "unexpected symbol");
  }

  static void Range (IntSet followers) {
  // Range == "[" OneRange { OneRange } "]" .
    accept(LBrackSym, "[ expected");
    OneRange(followers.union(new IntSet(RBrackSym)));
    while (FirstOneRange.contains(sym.kind))
      OneRange(followers.union(new IntSet(RBrackSym)));
    accept(RBrackSym, "] expected");
  }

  static void OneRange (IntSet followers) {
  // OneRange == Atom [ "-" Atom ] .
    Atom(followers.union(new IntSet(HyphenSym)));
    if (sym.kind == HyphenSym) { getSym(); Atom(followers); }
  }

  static void Atom (IntSet followers) {
  // Atom = atomic | escaped .
  // Note the use of a default option to mop up errors
    switch (sym.kind) {
      case AtomSym :     getSym(); break;
      case EscapedSym :  getSym(); break;
      default :          reportError("Invalid Atom"); break;
      }
   }

....

    getChar();                      // Lookahead character
    getSym();                       // Lookahead symbol
    RE(new IntSet(EOFSym));         // Start to parse from the goal symbol
    if (!errors) System.out.println("Parsed correctly");
```