Computer Science 3 - 2015

Programming Language Translation

Practical 5 for week beginning 28 September 2015 - 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 `PRAC5A.ZIP`.

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, 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 * and / or *** internally */`).

```
static int LiteralKind(StringBuilder lex) {
  string s = lex.ToString();              // convert it only once!
  if (s.equals("int")) return intSym;
  if (s.equals("char")) return charSym;
  if (s.equals("bool")) return boolSym;
  if (s.equals("void")) return voidSym;
  return identSym;
}

static boolean isIdentCh(char ch) {       // _allow_for_under_scores
  return Character.isLetter(ch) || (ch == '_');
}

static void GetSym() {
// Scans for next sym from input
  while (ch > EOF && ch <= ' ') GetChar();
  if (ch == '/') {                        // might (or might not) be a comment
    GetChar();
    if (ch == '*') {                      /* style comments */
      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;
    } else
    if (ch == '/') {                      // style comments
      do GetChar(); while (ch != '\n' && ch != EOF);
      if (ch == EOF)
        ReportError("unclosed comment");  // next sym would be EOFSym
      GetChar(); GetSym(); return;
    }
    else {                                // slash appeared on its own - not a comment
      sym = new Token(noSym, "/"); return;
    }
  }
  else {
    StringBuilder symLex = new StringBuilder();
    int symKind;
    if (isIdentCh(ch)) {
      do {
```

```
        symLex.Append(ch); GetChar();
      } while (isIdentCh(ch) || Character.isDigit(ch));
      symKind = literalKind(symLex);
    }
    else if (Character.isDigit(ch)) {
      do {
        symLex.Append(ch); GetChar();
      } while (Character.isDigit(ch));
      symKind = numSym;
    }
    else {
      symLex.Append(ch);                      // do this once!
      switch (ch) {
        case EOF:
          symLex = new StringBuilder("EOF");   // special representation
          symKind = EOFSym; break;             // no need to GetChar here, of course
        case ';':
          symKind = semicolonSym; GetChar(); break;
        case ',':
          symKind = commaSym; GetChar(); break;
        case '(':
          symKind = lParenSym; GetChar(); break;
        case '[':
          symKind = lBrackSym; GetChar(); break;
        case ')':
          symKind = rParenSym; GetChar(); break;
        case ']':
          symKind = rBrackSym; GetChar(); break;
        case '*':
          symKind = pointerSym; 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 FirstCdecls   = new IntSet(intSym, charSym, boolSym, voidSym, EOFSym);
static IntSet FirstDecList  = new IntSet(intSym, charSym, boolSym, voidSym);
static IntSet FirstType     = new IntSet(intSym, charSym, boolSym, voidSym);
static IntSet FirstOneDecl  = new IntSet(pointerSym, identSym, lParenSym);
static IntSet FirstDirect   = new IntSet(identSym, lParenSym);
static IntSet FirstSuffix   = new IntSet(lBrackSym, lParenSym);
static IntSet FirstParams   = new IntSet(lParenSym);
static IntSet FirstOneParam = new IntSet(intSym, charSym, boolSym, voidSym);
static IntSet FirstArray    = new IntSet(lBrackSym);

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 Cdecls() {
// Cdecls = { DecList } EOF .
  while (FirstDecList.Contains(sym.kind)) DecList();
  Accept(EOFSym, "EOF expected");
} // CDecls

static void DecList() {
// DecList = Type OneDecl { "," OneDecl } ";"  .
  Type();
  OneDecl();
  while (sym.kind == commaSym) {
    GetSym(); OneDecl();
  }
  Accept(semicolonSym, "semicolon expected");
}

static void Type() {
// Type = "int" | "void" | "bool" | "char" .
  if (FirstType.Contains(sym.kind)) GetSym();
  else Abort("invalid Type");
```

```
}

static void OneDecl() {
// OneDecl = "*" OneDecl | Direct .
  if (sym.kind == pointerSym) {
    GetSym(); OneDecl();
  }
  else Direct();
}

static void Direct() {
// Direct = ( ident | "(" OneDecl ")" ) [ Suffix ] .
  switch(sym.kind) {
    case identSym:
      GetSym(); break;
    case lParenSym:
      GetSym(); OneDecl(); Accept(rParenSym, ") expected"); break;
    default: Abort("invalid start to Direct"); break;
  }
  if (FirstSuffix.Contains(sym.kind)) Suffix();
}

static void Suffix() {
// Suffix = Array { Array } | Params .
  switch(sym.kind) {
    case lBrackSym:
      Array();
      while (sym.kind == lBrackSym) Array();
      break;
    case lParenSym:
      Params();
      break;
    default:
      Abort("invalid start to suffix"); break;
  }
}

static void Params() {
// Params = "(" [ OneParam { "," OneParam } ] ")" .
  Accept(lParenSym, "( expected");
  if (FirstOneParam.Contains(sym.kind)) {
    OneParam();
    while (sym.kind == commaSym) {
      GetSym(); OneParam();
    }
  }
  Accept(rParenSym, ") expected");
}

static void OneParam() {
// OneParam = Type [ OneDecl ] .
  Type();
  if (FirstOneDecl.Contains(sym.kind)) OneDecl();
}

static void Array() {
// Array = "[" [ number ] "]" .
  Accept(lBrackSym, "[ expected");
  if (sym.kind == numSym) GetSym();
  Accept(rBrackSym, "] 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. To do this one needs to know the appropriate `FIRST` and `FOLLOW` sets. A problem with that approach is that the sets become larger than one would like. The approach below shows how we can start with the followers of `CDecls` and then compute the additions to the follower set that we pass as a parameter to subsequent routines. But this is overly clumsy (read on for the next exciting instalment!)

```
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();
}

static void Cdecls(IntSet followers) {
// Cdecls = { DecList } EOF .
// First Test allows us to skip any real rubbish at the beginning
  Test(followers.Union(FirstCdecls), new IntSet(), "bad start to file");
  while (FirstDecList.Contains(sym.kind))
    DecList(followers.Union(FirstDecList).Union(new IntSet(EOFSym)));
  Accept(EOFSym, "EOF expected");
}

static void DecList(IntSet followers) {
// DecList = Type OneDecl { "," OneDecl } ";"  .
  Test(FirstDecList, followers, "invalid start to DecList");
  if (FirstDecList.Contains(sym.kind)) {
    Type(followers.Union(FirstOneDecl));
    OneDecl(followers.Union(new IntSet(commaSym, semicolonSym)));
    while (sym.kind == commaSym) {
      GetSym();
      OneDecl(followers.Union(new IntSet(commaSym, semicolonSym)));
    }
    Accept(semicolonSym, "semicolon expected");
  }
  Test(followers, new IntSet(), "DecList : unexpected symbol");
}

static void Type(IntSet followers) {
// Type = "int" | "void" | "bool" | "char" .
  Test(FirstType, followers, "invalid start to Type");
  if (FirstType.Contains(sym.kind)) GetSym();
  Test(followers, new IntSet(), "Type : unexpected symbol");
}

static void OneDecl(IntSet followers) {
// OneDecl = "*" OneDecl | Direct .
  Test(FirstOneDecl, followers, "invalid start to OneDecl");
  if (FirstOneDecl.Contains(sym.kind)) {
    if (sym.kind == pointerSym) {
      GetSym(); OneDecl(followers);
    }
    else Direct(followers);
  }
  Test(followers, new IntSet(), "OneDecl : unexpected symbol");
}

static void Direct(IntSet followers) {
// Direct = ( ident | "(" OneDecl ")" ) [ Suffix ] .
  Test(FirstDirect, followers.Union(FirstSuffix), "invalid start to Direct");
  if (FirstDirect.Contains(sym.kind))
    switch(sym.kind) {
      case identSym:
        GetSym(); break;
      case lParenSym:
        GetSym();
        OneDecl(followers.Union(new IntSet(rParenSym)));
        Accept(rParenSym, ") expected"); break;
      default: break; // already handled
    }
  if (FirstSuffix.Contains(sym.kind)) Suffix(followers);
  Test(followers, new IntSet(), "Direct : unexpected symbol");
}

static void Suffix(IntSet followers) {
// Suffix = Array { Array } | Params .
  Test(FirstSuffix, followers, "invalid start to Suffix");
  if (FirstArray.Contains(sym.kind)) {
    Array(followers.Union(FirstArray));
```

```
    while (FirstArray.Contains(sym.kind)) Array(followers.Union(FirstArray));
  }
  else
    if (FirstParams.Contains(sym.kind)) Params(followers);
  Test(followers, new IntSet(), "Suffix : unexpected symbol");
}

static void Params(IntSet followers) {
// Params = "(" [ OneParam { "," OneParam } ] ")" .
  Test(FirstParams, followers, "invalid start to Params");
  if (FirstParams.Contains(sym.kind)) {
    Accept(lParenSym, "( expected");
    if (FirstOneParam.Contains(sym.kind)) {
      OneParam(followers.Union(new IntSet(commaSym, rParenSym)));
      while (sym.kind == commaSym) {
        GetSym();
        OneParam(followers.Union(new IntSet(commaSym, rParenSym)));
      }
    }
  }
  Accept(rParenSym, ") expected");
  Test(followers, new IntSet(), "Params : unexpected symbol");
}

static void OneParam(IntSet followers) {
// OneParam = Type [ OneDecl ] .
  Test(FirstOneParam, followers.Union(FirstOneDecl), "invalid start to OneParam");
  if (FirstType.Contains(sym.kind)) Type(followers.Union(FirstOneDecl));
  if (FirstOneDecl.Contains(sym.kind)) OneDecl(followers);
  Test(followers, new IntSet(), "OneParam : unexpected symbol");
}

static void Array(IntSet followers) {
// Array = "[" [ number ] "]" .
  Test(FirstArray, followers, "invalid start to Array");
  if (FirstArray.Contains(sym.kind)) {
    Accept(lBrackSym, "[ expected");
    if (sym.kind == numSym) GetSym();
    Accept(rBrackSym, "] expected");
  }
  Test(followers, new IntSet(), "Array : unexpected symbol");
}
```

When one looks at the `FIRST` sets one should be struck by the fact that many of them are the same. After a bit of reflection on all this, one should be able to see that we can actually economise on (possibly slow and expensive) calls to the Test function, simply by incorporating them only at certain critical points. The complete solution below shows how I suggest this problem might be solved. It is a bit crude in its positioning of error messages.

```
static void Cdecls(IntSet followers) {
// Cdecls = { DecList } EOF .
// First Test allows us to skip any real rubbish at the beginning
  Test(followers.Union(FirstCdecls), new IntSet(), "bad start to declarations");
  while (FirstDecList.Contains(sym.kind))
    DecList(followers.Union(FirstDecList).Union(new IntSet(EOFSym)));
  Accept(EOFSym, "EOF expected");
}

static void DecList(IntSet followers) {
// DecList = Type OneDecl { "," OneDecl } ";"  .
  Type(followers.Union(FirstOneDecl));
  OneDecl(followers.Union(new IntSet(commaSym, semicolonSym)));
  while (sym.kind == commaSym) {
    GetSym();
    OneDecl(followers.Union(new IntSet(commaSym, semicolonSym)));
  }
  Test(new IntSet(semicolonSym), followers, "semicolon expected");
  if (sym.kind == semicolonSym) GetSym();
}

static void Type(IntSet followers) {
// Type = "int" | "void" | "bool" | "char" .
  if (FirstType.Contains(sym.kind)) GetSym(); else ReportError("invalid Type");
}

static void OneDecl(IntSet followers) {
// OneDecl = "*" OneDecl | Direct .
  Test(FirstOneDecl, followers, "invalid Declarations");
  if (FirstOneDecl.Contains(sym.kind)) {
    if (sym.kind == pointerSym) {
```

```
      GetSym(); OneDecl(followers);
    }
    else Direct(followers);
  }
  Test(followers, new IntSet(), "unexpected symbol");
}

static void Direct(IntSet followers) {
// Direct = ( ident | "(" OneDecl ")" ) [ Suffix ] .
  switch(sym.kind) {
    case identSym:
      GetSym(); break;
    case lParenSym:
      GetSym();
      OneDecl(followers.Union(new IntSet(rParenSym)));
      Accept(rParenSym, ") expected"); break;
    default:
      ReportError("identifier or ( expected"); break;
    }
  if (FirstSuffix.Contains(sym.kind)) Suffix(followers);
  Test(followers, new IntSet(), "unexpected symbol");
}

static void Suffix(IntSet followers) {
// Suffix = Array { Array } | Params .
  if (FirstArray.Contains(sym.kind)) {
    Array(followers.Union(FirstArray));
    while (FirstArray.Contains(sym.kind)) Array(followers.Union(FirstArray));
  }
  else Params(followers);
}

static void Params(IntSet followers) {
// Params = "(" [ OneParam { "," OneParam } ] ")" .
  Accept(lParenSym, "( expected");
  if (FirstOneParam.Contains(sym.kind)) {
    OneParam(followers.Union(new IntSet(commaSym, rParenSym)));
    while (sym.kind == commaSym) {
      GetSym();
      OneParam(followers.Union(new IntSet(commaSym, rParenSym)));
    }
  }
  Accept(rParenSym, ") expected");
}

static void OneParam(IntSet followers) {
// OneParam = Type [ OneDecl ] .
  Type(followers.Union(FirstOneDecl));
  if (FirstOneDecl.Contains(sym.kind)) OneDecl(followers);
  Test(followers, new IntSet(), "unexpected symbol in parameter declaration");
}

static void Array(IntSet followers) {
// Array = "[" [ number ] "]" .
  Accept(lBrackSym, "[ expected");
  if (sym.kind == numSym) GetSym();
  Accept(rBrackSym, "] expected");
}
```

It could be improved still further by setting up a few more static sets, for example:

```
static IntSet puncSet0 = new IntSet(semicolonSym);
static IntSet puncSet1 = new IntSet(commaSym, semicolonSym);

static void DecList(IntSet followers) {
// DecList = Type OneDecl { "," OneDecl } ";"  .
  Type(followers.Union(FirstOneDecl));
  OneDecl(followers.Union(puncSet1));
  while (sym.kind == commaSym) {
    GetSym();
    OneDecl(followers.Union(puncSet1));
  }
  Test(puncSet0, followers, "semicolon expected");
  if (sym.kind == semicolonSym) GetSym();
}
```