Computer Science 3 - 2010

Programming Language Translation

Practical for Week 23, beginning 27 September 2010 - Solutions

This practical was done fairly well by all but a few groups, These parsers and scanners 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 should 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 (*)?

```
static int literalKind(StringBuilder lex) {
  String s = lex.toString();
  if (s.equals("ARRAY"))    return arraySym;
  if (s.equals("END"))      return endSym;
  if (s.equals("OF"))       return ofSym;
  if (s.equals("POINTER"))  return pointerSym;
  if (s.equals("RECORD"))   return recordSym;
  if (s.equals("SET"))      return setSym;
  if (s.equals("TO"))       return toSym;
  if (s.equals("TYPE"))     return typeSym;
  if (s.equals("VAR"))      return varSym;
  return identSym;
}

static void getSym() {
// Scans for next sym from input
  while (ch > EOF && ch <= ' ') getChar();
  StringBuilder symLex = new StringBuilder();
  int symKind = noSym;
  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 = semicolonSym; getChar(); break;
```

```
      case ':':
        symKind = colonSym; getChar(); break;

      case '(':  // might 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"); // sym will be EOFSym
          getChar(); getSym(); return;       // return is crucial here
        }
        else symKind = lparenSym; break;
      case ')':
        symKind = rparenSym; getChar(); break;
      case '[':
        symKind = lbrackSym; getChar(); break;
      case ']':
        symKind = rbrackSym; getChar(); break;
      case '=':
        symKind = equalSym; getChar(); break;
      case '.':
        getChar();
        if (ch == '.') {
          symLex.append(ch);
          symKind = dotdotSym; getChar();
        }
        else symKind = periodSym;
        break;
      default :
        symKind = noSym; getChar(); break;
    }
  }
  sym = new Token(symKind, symLex.toString());
} // getSym
```

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

```
static IntSet
  FirstDeclaration = new IntSet(typeSym, varSym),
  EndDeclSyms      = new IntSet(EOFSym, semicolonSym);

static void Mod2Decl() {
// Mod2Decl    = { Declaration } EOF .
  while (FirstDeclaration.contains(sym.kind))
    Declaration();
  accept(EOFSym, "EOF expected");
}

static void Declaration() {
// Declaration =   "TYPE"  { TypeDecl  SYNC ";" }
//               | "VAR"   { VarDecl   SYNC ";" } .
  switch(sym.kind) {
    case typeSym :
      getSym();
      while (sym.kind == identSym) {
        TypeDecl();
        accept(semicolonSym, "; expected");
      }
      break;
    case varSym :
      getSym();
      while (sym.kind == identSym) {
        VarDecl();
        accept(semicolonSym, "; expected");
      }
      break;
    default:
      abort("unrecognizable declaration - TYPE or VAR expected");
      break;
  }
}
```

```
static void TypeDecl() {
// TypeDecl = identifier "=" Type .
  accept(identSym, "identifier expected");  // getSym() would be dangerous
  accept(equalSym, "= expected");
  Type();
}

static void VarDecl() {
// VarDecl = IdentList ":" Type .
  IdentList();
  accept(colonSym, ": expected");
  Type();
}

static void Type() {
// Type = SimpleType | ArrayType | RecordType | SetType | PointerType .
  switch (sym.kind) {
    case identSym:
    case lparenSym:
    case lbrackSym:
      SimpleType(); break;
    case arraySym:
      ArrayType(); break;
    case recordSym:
      RecordType(); break;
    case setSym:
      SetType(); break;
    case pointerSym:
      PointerType(); break;
    default:
      abort("invalid start to Type");
      break;
  }
}

static void SimpleType() {
// SimpleType = QualIdent [ Subrange ] | Enumeration | Subrange .
  switch (sym.kind) {
    case identSym:
      QualIdent();
      if (sym.kind == lbrackSym) Subrange();
      break;
    case lparenSym:
      Enumeration();
      break;
    case lbrackSym:
      Subrange();
      break;
    default:
      abort("invalid start to SimpleType");
      break;
  }
}

static void QualIdent() {
// QualIdent = identifier { "." identifier } .
  accept(identSym, "identifier expected");
  while (sym.kind == periodSym) {
    getSym();
    accept(identSym, "identifier expected");
  }
}

static void Subrange() {
// Subrange = "[" Constant ".." Constant "]"  .
  accept(lbrackSym, "[ expected");
  Constant();
  accept(dotdotSym, "] expected");
  Constant();
  accept(rbrackSym, "] expected");
}

static void Enumeration() {
// Enumeration = "(" IdentList ")" .
  accept(lparenSym, "( expected");          // getSym() would be dangerous
  IdentList();
  accept(rparenSym, ") expected");
}
```

```
static void Constant() {
// Constant = number | identifier .
  switch (sym.kind) {
    case numSym   :
      getSym(); break;
    case identSym :
      getSym(); break;
    default :
      abort("Illegal constant"); break;
  }
}

static void IdentList() {
// IdentList = identifier { "," identifier } .
  accept(identSym, "identifier expected");
  while (sym.kind == commaSym) {
    getSym();
    accept(identSym, "identifier expected");
  }
}

static void ArrayType() {
// ArrayType = "ARRAY" SimpleType { "," SimpleType } "OF" Type.
  accept(arraySym, "ARRAY expected");       // getSym() would be dangerous
  SimpleType();
  while (sym.kind == commaSym) {
    getSym();
    SimpleType();
  }
  accept(ofSym, "OF expected");
  Type();
}

static void RecordType() {
// RecordType = "RECORD" FieldLists "END" .
  accept(recordSym, "RECORD expected");     // getSym() would be dangerous
  FieldLists();
  accept(endSym, "END expected");
}

static void FieldLists() {
// FieldLists = FieldList { ";" FieldList } .
  FieldList();
  while (sym.kind == semicolonSym) {
    getSym();
    FieldList();
  }
}

static void FieldList() {
// FieldList = [ IdentList ":" Type ] .
  if (sym.kind == identSym) {
    IdentList();
    accept(colonSym, ": expected");
    Type();
  }
}

static void SetType() {
// SetType = "SET" "OF" SimpleType .
  accept(setSym, "SET expected");           // getSym() would be dangerous
  accept(ofSym, "OF expected");
  SimpleType();
}

static void PointerType() {
// PointerType = "POINTER" "TO" Type .
  accept(pointerSym, "POINTER expected");   // getSym() would be dangerous
  accept(toSym, "TO expected");
  Type();
}
```

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 like the code below, you had better come to ask!

```
static void Constant() {
// Constant = number | identifier .
  if (sym.kind == numSym || sym.kind == identSym) getSym();
}
```

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 .
```

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. These points are marked "dangerous" in the solution above.

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 `Declaration()` parser, which attempts some simple synchronisation.

```
static void synchOnSemicolon() {
// Simple attempt to synchronize on semicolon or EOF
  if (sym.kind != semicolonSym) {
    reportError("; expected");
    do getSym(); while (!EndDeclSyms.contains(sym.kind));
  }
  getSym();
}

static void Declaration() {
// Declaration =   "TYPE"  { TypeDecl  SYNC ";" }
//               | "VAR"   { VarDecl   SYNC ";" } .
  switch(sym.kind) {
    case typeSym :
      getSym();
      while (sym.kind == identSym) {
        TypeDecl();
        synchOnSemicolon();
      }
      break;
    case varSym :
      getSym();
      while (sym.kind == identSym) {
        VarDecl();
        synchOnSemicolon();
      }
      break;
    default:
      abort("unrecognizable declaration - TYPE or VAR expected");
      break;
  }
}
```

although to get this to work better one would probably not abort parsing on other errors.

Finally, notice that in some (other) applications it might be 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:

```
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);
}
```