Computer Science 3 - 2017

Programming Language Translation

Practical 5 for week beginning 14 August 2017 - 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 is available on the WWW pages in the file `PRAC5A.ZIP`.

The scanner was not well done by some people. I have commented in great detail on how the extensions for Task 4 could have been done. A Fortran comment is, effectively, comprised of an EOL followed immediately by a C and then by printable characters up to but not including the next EOL. This gives a simple way of recognizing them, but, as most discovered, one has to do something special to handle the very real possibility that the very first real character in the source file is a C (which is, of course, not preceded by anything, let alone an EOL). But that is easy to fix (there is always a neat simple solution to a Pat Terry problem). The scanner assumes that a character has already been read before `GetSym()` is called for the first time. Rather than use `GetChar()` to do that, as in the skeleton kit, just assign EOL to `ch` and fool the system that way. It won't cause any problem if there is no introductory comment either, when you think about it. So think about it.

There were some very tortuous ways of recognising a string literal with embedded double quotes. Once again, try to keep it as simple as you can. But no simpler! As with the .XXX. tokens, there is a finite chance that a user will not terminate a string before reaching an EOL (or before an EOF in freak cases). So the code has to be prepared to deal with that as well.

Incidentally, I think most text editors probably add an LF or a CRLF at the end of the last line of text -meaning that one should always be able to detect the last end of line before detecting the pseudo EOF. But not all do, hence the need to check both.

Comments are simple conceptually, but awkward to define and even to ignore in practice, especially if you can nest them, which some languages allow you to do (a very useful feature, but a dangerous one).

```
    const int
      noSym        =  0,
      numSym       =  1,
      identSym     =  2,
      equalSym     =  2,
      assignSym    =  3,
      ...  // see full solution

    static int LiteralKind(StringBuilder lex) {
      string str = lex.ToString().ToUpper();
      switch (str) {
        case "CONTINUE" : return continueSym;
        case "DO" :       return doSym;
        case "ELSE" :     return elseSym;
        ...
        default :         if (str.Length > 6) return noSym;
                          else return identSym;
      }
    } // LiteralKind
    static int OperatorKind(StringBuilder lex) {
      switch (lex.ToString().ToUpper()) {
        case ".EQ." :     return eqSym;
        case ".NE." :     return neSym;
        ...
        default :         return noSym;
      }
    } // OperatorKind

    static bool IsStringChar(char ch) {
      return ch >= ' ' && ch != '\'';
    } // IsStringChar

    static void GetSym() {
    // Scans for next sym from input
    // To deal with comments we arrange that before this is called for the very first time
    // ch is initialised to EOL.  The if the first line is a comment it is picked up easily
    // This is a trick that suits Fortran - one might not normally do it
      while (ch > EOF && ch <= ' ' && ch != EOL) GetChar();
      StringBuilder symLex = new StringBuilder();
      int symKind = noSym;
      if (Char.IsLetter(ch)) {                  // identifier or key word
        do {
          symLex.Append(ch); GetChar();
        } while (Char.IsLetterOrDigit(ch));
        symKind = LiteralKind(symLex);
      }
      else if (Char.IsDigit(ch)) {              // integer number
        do {
          symLex.Append(ch); GetChar();
        } while (Char.IsDigit(ch));
        symKind = numSym;
      }
      else {                                    // single character tokens , or ones that start
                                                // with distinctive single characters
        symLex.Append(ch);
        switch (ch) {
          case EOL :
            symLex = new StringBuilder("EOL");   // special representation for EOL for debugging
            symKind = EOLSym; GetChar();
            if (Char.ToUpper(ch) == 'C') {       // EOL + C signifies a comment
              do {                               // retain the text of the comment for test purposes
                symLex.Append(ch); GetChar();    // but treat it as an EOLsym which is significant
              } while (ch != EOL && ch != EOF);  // and beware of a rogue EOF
            }
            break;
          case EOF:
            symLex = new StringBuilder("EOF");   // special representation for EOF for debugging
            symKind = EOFSym;                    // No need to GetChar here, of course
            break;
          case '.' :                             // .op.  operators and logical constants
            GetChar();
            while (Char.IsLetter(ch)) {
              symLex.Append(ch); GetChar();
            }
            if (ch == '.') {                     // if it isn't,   .xxx   must be returned as a noSym
              symLex.Append(ch); GetChar();
            }
            symKind = OperatorKind(symLex);      // check for a valid   .xxxxx.   token
            break;
          case '\'' :                            // start of string literal detected
            while (true) {                       // a rare case where    while (true)   gives a neat way
              GetChar();
              while (IsStringChar(ch)) {         // will stop at next ' or EOL or EOF
                symLex.Append(ch); GetChar();
              }
              if (ch != '\'') break;             // abort if it runs off end of line or file prematurely
              symLex.Append(ch); GetChar();      // look at character after the '
              if (ch != '\'') {                  // not a quote means we have reached the end of string
                symKind = stringSym;
                break;
              }
              symLex.Append(ch);                 // no, we found a double quote - keep going round loop
            }
            break;
          case '(' :                             // the rest are easy, but note the GetChar() calls
            symKind = lparenSym; GetChar();
            break;
          case ')' :
            symKind = rparenSym; GetChar();
            break;
          case '+' :
            symKind = addSym; GetChar();
            break;
          case '-' :
            symKind = subSym; GetChar();
            break;
          case '*' :
            symKind = mulSym; GetChar();
            break;
          case '/' :
            symKind = divSym; GetChar();
            break;
          case '=' :
            symKind = assignSym; GetChar();
            break;
          case ',' :
            symKind = commaSym; GetChar();
            break;
          default :
            symKind = noSym; GetChar();
            break;
        }
      }
      sym = new Token(symKind, symLex.ToString());
    } // GetSym
```

Below is most of a simple "sudden death" parser, devoid of error recovery. I have commented all the `Accept()` and `GetSym()` calls to indicate why each is used in preference to the other one.

Note the `default` clauses in the `switch` statements.

```
    static IntSet
      FirstVars = new IntSet(logicalSym, integerSym),
      RelopSyms = new IntSet(ltSym, leSym, gtSym, geSym, eqSym, neSym ),
      AddopSyms = new IntSet(addSym, subSym),
      MulopSyms = new IntSet(mulSym, divSym),
      ConstSyms = new IntSet(trueSym, falseSym, numSym),
      FirstPrn  = new IntSet(printSym, printOSym),
      FirstLim  = new IntSet(identSym, goSym, goToSym, readSym, printSym, printOSym, stopSym, continueSym),
      FirstGen  = new IntSet(ifSym, doSym).Union(FirstLim);
      // It may be safer just to enumerate them all - the Library is a bit dodgy on sets 2017/08/26

           // One of the reasons for using Accept methods is that it gives a uniform way of reporting on errors
           // If you want to change that way, there is the one common place to do it, not a whole lot of
           // changes needed all over the code.

           // Note that I have used an unguarded "GetSym" when it is obviously quite safe to do so (because the
           // test for an acceptable token has already been done).

    static void Accept(int wantedSym, string errorMessage) {
    // Gets the next token if the current token matches   wantedSym
      if (sym.kind == wantedSym) GetSym(); else Abort(errorMessage);
    } // Accept

    static void Accept(IntSet allowedSet, string errorMessage) {
    // Gets the next token if the current token matches an element of   allowedSet
      if (allowedSet.Contains(sym.kind)) GetSym(); else Abort(errorMessage);
    } // Accept

    static void Program() {                                   // correction to grammar as originally
                                                              // supplied
    //  Program = { eol } "PROGRAM" Ident EOLS
    //            { VarDeclarations }
    //            { GeneralStatement }
    //            "END" WEAK eol { eol } EOF .                // can have eols, comments after END
                                                              // but no labels.  Best to check for EOF

      while (sym.kind == EOLSym) GetSym();                    // No need for Accept
      Accept(programSym, " Program expected");                // GetSym alone would be dangerous
      Ident();
      EOLS();
      while (FirstVars.Contains(sym.kind)) VarDeclarations();
      while (FirstGen.Contains(sym.kind))  GeneralStatement();
      Accept(endSym, " END expected");                        // GetSym alone would be dangerous
      if (sym.kind == EOLSym) {                               // be generous - the EOL is a WEAK token
        GetSym(); while (sym.kind == EOLSym) GetSym();        // No need for Accept
      }
      Accept(EOFSym, " EOF expected");
    } // Program
```

```
static void EOLS() {
//  EOLS = SYNC eol { eol } [ Label ] .
  Accept(EOLSym, " end of line expected");                    // GetSym alone would be dangerous
  while (sym.kind == EOLSym) GetSym();                        // No need for Accept
  if (sym.kind == numSym) Label();
} // EOLS

static void VarDeclarations() {
//  VarDeclarations = ( "INTEGER" | "LOGICAL" ) OneVar { WEAK "," OneVar } EOLS  .
  Accept(FirstVars, "invalid start to varDeclarations");     // GetSym alone would be dangerous
  OneVar();
  while (sym.kind == commaSym) {
    GetSym(); OneVar();                                       // No need for Accept
  }
  EOLS();
} // VarDeclarations

static void OneVar() {
//  OneVar = Ident [ "(" IntConst ")" ] .
  Accept(identSym, " identifier expected");                   // GetSym alone would be dangerous
  if (sym.kind == lparenSym) {
    GetSym(); IntConst();                                     // No need for Accept
    Accept(rparenSym, " ) expected");                         // GetSym alone would be dangerous
  }
} // OneVar

static void GeneralStatement() {
//  Some statements (IF, DO) are not permitted within a logical IF statement */
//  GeneralStatement = LimitedStatement | IfStatement | DoStatement  .
  switch (sym.kind) {
    case ifSym :
      IfStatement(); break;
    case doSym :
      DoStatement(); break;
    default:
      LimitedStatement(); break;                              // catch bad statements later
  }
} // GeneralStatement

static void LimitedStatement() {
//  Only a few statements are permitted as the subsidiary within a logical IF statement
//  LimitedStatement
//  = SYNC (   Assignment
//           | GoToStatement
//           | ReadStatement
//           | PrintStatement
//           | "STOP"
//           | "CONTINUE"
//         ) EOLS .
  switch(sym.kind) {
    case identSym :
       Assignment();    break;
    case goSym :                                              // two forms of this statement
    case goToSym :
      GoToStatement();  break;
    case readSym :
      ReadStatement();  break;
    case printSym :                                           // two forms of this statement
    case printOSym:
      PrintStatement(); break;
    case stopSym :
    case continueSym :
      GetSym();         break;                                // No need for Accept
    default :
      Abort(" unrecognised statement"); break;                // Any bad statement will be caught here
  }
  EOLS();
} // LimitedStatement

static void Assignment() {
//  Assignment = Designator "=" Expression .
  Designator();
  Accept(assignSym, " = expected");                           // GetSym alone would be dangerous
  Expression();
} // Assignment
```

```
static void Designator() {
//  Designator = Ident [  "(" Expression ")" ] .
  Ident();
  if (sym.kind == lparenSym) {
    GetSym(); Expression();                             // No need for Accept
    Accept(rparenSym, " ) expected");                   // GetSym alone would be dangerous
  }
} // Designator

static void IfStatement() {                             // slight rearrangement of production
//  IfStatement = "IF" "(" Expression ")"
//                (    Label "," Label "," Label EOLS
//                  |  "THEN"  EOLS { GeneralStatement }
//                       [ "ELSE"  EOLS { GeneralStatement } ]
//                     "ENDIF" EOLS
//                  |  LimitedStatement
//                ) .
  Accept(ifSym, " IF expected");                        // GetSym alone might be dangerous
  Accept(lparenSym, " ( expected");                     // GetSym alone would be dangerous
  Expression();
  Accept(rparenSym, " ) expected");                     // GetSym alone would be dangerous
  switch (sym.kind) {
    case numSym :                                       // It is an Arithmetic IfStatement
      Label(); Accept(commaSym, " , expected");         // GetSym alone would be dangerous
      Label(); Accept(commaSym, " , expected");         // GetSym alone would be dangerous
      Label(); EOLS();
      break;
    case thenSym :                                      // It is a Block IfStatement
      GetSym();                                         // No need for Accept
      EOLS();
      while (FirstGen.Contains(sym.kind)) GeneralStatement();
      if (sym.kind == elseSym) {
        GetSym();                                       // No need for Accept
        EOLS();
        while (FirstGen.Contains(sym.kind)) GeneralStatement();
      }
      Accept(endifSym, " ENDIF expected");              // GetSym alone would be dangerous
      EOLS();
      break;
    default :
      LimitedStatement();                              // It should be a Logical IfStatement, but
      break;                                           // this will catch any bad if statements
  } // switch
} // IfStatement

static void DoStatement() {
//  DoStatement = "DO" Label [ "," ]
//               Ident "=" Expression "," Expression [ "," Expression ] EOLS .
  Accept(doSym, " DO expected");                        // GetSym alone might be dangerous
  Label();
  if (sym.kind == commaSym) GetSym();                   // No need for Accept
  Ident();
  Accept(assignSym, " = expected");                     // GetSym alone would be dangerous
  Expression();
  Accept(commaSym, " , expected");                      // GetSym alone would be dangerous
  Expression();
  if (sym.kind == commaSym) {
    GetSym(); Expression();                             // No need for Accept
  }
  EOLS();
} // DoStatement

static void GoToStatement() {
//  GoToStatement = ( "GOTO" | "GO" "TO" )
//                  (   Label
//                    | "(" Label { WEAK "," Label } ")" [ "," ] Expression
//                  ) .
  if (sym.kind == goSym) {
    GetSym(); Accept(toSym, " TO expected");            // GetSym alone would be dangerous
  }
  else Accept(goToSym, " GOTO expected");               // GetSym alone would be dangerous
  if (sym.kind == numSym) Label();                      // simple GoTo statement
  else {                                                // This form gives a switch statement
    Accept(lparenSym, " ( expected");                   // GetSym alone would be dangerous
    Label();
    while (sym.kind == commaSym) {
      GetSym(); Label();                                // No need for Accept
    }
    Accept(rparenSym, " ) expected");                   // GetSym alone would be dangerous
    if (sym.kind == commaSym) GetSym();                 // No need for Accept
    Expression();
    }
  } // GoToStatement

  static void ReadStatement() {
  //  ReadStatement = "READ" "*" { WEAK "," ReadElement } .
    Accept(readSym, " READ expected");                       // GetSym alone would be dangerous
    Accept(mulSym, " * expected");                           // GetSym alone would be dangerous
    while (sym.kind == commaSym) {
      GetSym(); ReadElement();                               // No need for Accept
    }
  } // ReadStatement

  static void ReadElement() {
  //  ReadElement = Designator .
    Designator();
  } // ReadElement

  static void PrintStatement() {
  //  PrintStatement = ( "PRINTO" | "PRINT" ) "*" { WEAK "," PrintElement } .
    Accept(FirstPrn, " PRINT or PRINTO expected");           // GetSym alone would be dangerous
    Accept(mulSym, " * expected");                           // GetSym alone would be dangerous
    while (sym.kind == commaSym) {
      GetSym(); PrintElement();                              // No need for Accept
    }
  } // PrintStatement

  static void PrintElement() {
  //  PrintElement = stringLit | Expression .
    if (sym.kind == stringSym) GetSym();                     // No need for Accept
    else Expression();                                       // Expression will catch any bad PrintElement
  } // PrintElement

  static void Expression() {
  //  Expression = AndExp { ".OR." AndExp } .
    AndExp();
    while (sym.kind == orSym) {
      GetSym(); AndExp();                                    // No need for Accept
    }
  } // Expression

  static void AndExp() {
  //  AndExp = NotExp { ".AND." NotExp } .
    NotExp();
    while (sym.kind == andSym) {
      GetSym(); NotExp();                                    // No need for Accept
    }
  } // AndExp

  static void NotExp() {
  //  NotExp = [ ".NOT." ] RelExp .
    if (sym.kind == notSym) GetSym();                        // No need for Accept
    RelExp();
  } // NotExp

  static void RelExp() {
  //  RelExp = AddExp [ RelOp AddExp ] .
    AddExp();
    if (RelopSyms.Contains(sym.kind)) {
      GetSym(); AddExp();                                    // No need for Accept
    }
  } // RelExp

  static void AddExp() {
  //  AddExp = [ AddOp ] MultExp { AddOp MultExp } .
    if (AddopSyms.Contains(sym.kind)) GetSym();              // No need for Accept
    MultExp();
    while (AddopSyms.Contains(sym.kind)) {
      GetSym(); MultExp();                                   // No need for Accept
    }
  } // AddExp

  static void MultExp() {
  //  MultExp = Factor { MulOp Factor } .
    Factor();
    while (MulopSyms.Contains(sym.kind)) {
      GetSym(); Factor();                                    // No need for Accept
    }
  } // MultExp
```

```
static void Factor() {
//  Factor  = Designator | Constant | "(" Expression ")" .
  if (sym.kind == identSym) Designator();
  else if (ConstSyms.Contains(sym.kind)) Constant();
  else if (sym.kind == lparenSym) {
    GetSym(); Expression();                              // No need for Accept
    Accept(rparenSym, " ) expected");                    // GetSym alone would be dangerous
  }
  else Abort(" invalid start to expression");            // any problems with expressions will
                                                         // eventually be caught here
} // Factor

static void Constant() {
//  Constant = IntConst | ".TRUE." | ".FALSE." .
  Accept(ConstSyms, " constant expected");               // GetSym alone would not work
} // Constant

static void AddOp() {
//  AddOp = "+" | "-" .
  Accept(AddopSyms, " + or - expected");                 // GetSym alone would not work
} // AddOp

static void MulOp() {
//  MulOp = "*" | "/" .
  Accept(MulopSyms, " * or / expected");                 // GetSym alone would not work
} // MulOp

static void RelOp() {
//  RelOp = ".LT." | ".LE." | ".GT." | ".GE." | ".EQ." | ".NE." .
  Accept(RelopSyms, " relational operator expected");    // GetSym alone would be dangerous
} // RelOp

static void Ident() {
//  Ident = identifier .
  Accept(identSym, " valid identifier expected");        // GetSym alone would be dangerous
} // Ident

static void IntConst() {
//  IntConst = number .
  Accept(numSym, " number expected");                    // GetSym alone would be dangerous
} // IntConst

static void Label() {
//  Label = number .
  Accept(numSym, " invalid label");                      // GetSym alone would be dangerous
} // Label
```

One of the class made a good suggestion about using a command line parameter to steer the program between doing a scanner-only checker and a complete parser. I'll remember that for future years if I teach the course again, but here is the sort of thing that one might do:

```
// +++++++++++++++++++++ Main driver function ++++++++++++++++++++++++++++++

public static void Main(string[] args) {
  // Open input and output files from command line arguments
  if (args.Length == 0) {
    Console.WriteLine("Usage: ToyFortran FileName [-testScanner] ");
    System.Environment.Exit(1);
  }
  input  = new InFile(args[0]);                        // should really test for presence of file!
  output = new OutFile(NewFileName(args[0], ".out"));
  bool testScanner = args.Length > 1;                  // Suggestion from student - well done!

  ch = '\n';                                           // EOL for the lookahead character - see notes above.
  if (testScanner)                                     // Useful to be able to do only this, even
                                                       // when working on the full parser
    do {
      GetSym();                                        // Lookahead token
      OutFile.StdOut.Write(sym.kind, 3);
      OutFile.StdOut.WriteLine(" " + sym.val);         // See what we got
    } while (sym.kind != EOFSym);
```

```
      else {
        GetSym();                                       // Lookahead symbol
        Program();                                      // Start to parse from the goal symbol
                   // if we get back here everything must have been satisfactory
        Console.WriteLine("Parsed correctly");
      } // testing full parser

      output.Close();

    } // Main
```

A point to make is that, for safety, 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();
        } // Something
```

and not as

```
        void Something() {
          GetSym();
          SomethingElse();
        } // Something
```

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();
          }
        } // Something
```

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");
        } // 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;
  }
} // Something
```

although you could almost "get away" with

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

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 that all your `switch` statements *always* have a `default` clause (here or in other code you write).

However, this is an opportune time to comment further on pragmatic error detection. There is a fine balance between putting in too many checks and not enough checks. Consider the *Expression* parser as an example. It would be possible (and maybe thought a good idea) to check at the start of the *Expression* production that *sym* is in the appropriate FIRST set, and to do the same as the start of the production for *AndExp*, the production for *NotExp*, and so on all the way to the production for *Factor*. But all these first sets are much the same. So not doing any checking until one starts on the options for *Factor* works quite well enough - any *Expression* eventually sees a *Factor* being parsed. The same goes for statements - if one uses *LimitedStatement* as the "default" in the switch within *GeneralStatement* one can delay the checking that each and every statement starts correctly until the point is reached where *LimitedStatement* is called.