Computer Science 3 - 2017

Programming Language Translation

Practical 5 for week beginning 14 August 2017

Hand in your final solutions to this practical *before* lunch time on your next practical day, correctly packaged in a transparent folder with your cover sheets. Please do NOT come to a practical and spend the first hour printing or completing solutions from the previous week's exercises. Since the practical will have been done on a group basis, please hand in one copy of the cover sheet for each member of the group. These will be returned to you in due course, signed by the marker. **Please make it clear whose folder you have used for the electronic submission, for example g15A1234.** Lastly, please resist the temptation to carve up the practical, with each group member only doing one task. The group experience is best when you discuss each task together.

Objectives:

In this practical you are to

- develop a recursive descent parser and associated *ad hoc* scanner "from scratch" that will parse programs written in a language that is close to being a subset of the famous language FORTRAN 66.

Outcomes:

When you have completed this practical you should understand

- the inner workings of an *ad hoc* scanner;
- the inner workings of a recursive descent parser;
- how to test that a scanner and parser behave correctly;
- (hopefully) some aspects of a FORTRAN-like language.

To hand in:

This week you are required to hand in, besides the cover sheets (one per group member, please!):

- A listing of the final version of your source program, and some listings of input and output files;
- Electronic copies of the sources of your program.

Keep the prac sheet and your solutions until the end of the semester. Check carefully that your mark has been entered into the Departmental Records.

You are referred to the rules for practical submission which are clearly stated in our Departmental Handbook. However, for this course pracs must be posted in the "hand-in" box outside the laboratory and not given to demonstrators.

A rule not stated there, but which should be obvious, is that you are not allowed to hand in another group's or student's work as your own. Attempts to do this will result in (at best) a mark of zero and (at worst) severe disciplinary action and the loss of your DP. You are allowed - even encouraged - to work and study with other students, but if you do this you are asked to acknowledge that you have done so. You are expected to be familiar with the University Policy on Plagiarism, which you can consult on the Rhodes website.

WARNING. This exercise really requires you to do some real thinking and planning. Please do not just sit at a computer and hack away as most of you are wont to do. Sit in your groups and discuss your ideas with one another and with the demonstrators. If you don't do this you will probably find that the whole exercise turns into a nightmare, and I don't want that to happen.

Task 1 - a trivial task

Unpack the prac kit `PRAC5.ZIP`. In it you will find the skeleton of a system adapted for intermediate testing of a scanner (and to which you will later add a parser), and some simple test data files - but you really need to learn to develop your own test data.

Task 2 - get to grips with the problem.

Here is a grammar for most of our pseudo-Fortran subset:

```
COMPILER ToyFortran
/* ToyFortran level 1 grammar  - Coco/R for C#
   P.D. Terry, Rhodes University, 2017 */

IGNORECASE

CHARACTERS
  lf         = CHR(10) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  stringCh   = ANY - "'" - control .
  printable  = ANY - control .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  stringLit  = "'" { stringCh } "'" .
  eol        = lf .

IGNORE CHR(0) .. CHR(9) + CHR(11) .. CHR(31)

PRODUCTIONS
  ToyFortran        = { eol } "PROGRAM" Ident EOLS
                      { VarDeclarations }
                      { GeneralStatement }
                      "END" EOLS .
  EOLS              = SYNC eol { eol } [ Label ] .
  VarDeclarations   = ( "INTEGER" | "LOGICAL" ) OneVar { WEAK "," OneVar } EOLS  .
  OneVar            = Ident [ "(" IntConst ")" ] .
  GeneralStatement    /* Some statements are not permitted within a logical IF statement */
                    = LimitedStatement | IfStatement | DoStatement  .
  LimitedStatement    /* These are all permitted as the subsidiary within a logical IF statement */
                    = SYNC (   Assignment
                             | GoToStatement
                             | ReadStatement
                             | PrintStatement
                             | "STOP"
                             | "CONTINUE"
                           ) EOLS .
  Assignment        = Designator "=" Expression .
  Designator        = Ident [  "(" Expression ")" ] .
  IfStatement       = "IF" "(" Expression ")"
                       (    Label "," Label "," Label EOLS
                          |  (   LimitedStatement
                               |   "THEN"  EOLS { GeneralStatement }
                                 [ "ELSE"  EOLS { GeneralStatement } ]
                                   "ENDIF" EOLS
                             )
                       ) .
  DoStatement       = "DO" Label [ "," ]
                       Ident "=" Expression "," Expression [ "," Expression ] EOLS .
  GoToStatement     = ( "GOTO" | "GO" "TO" )
                        (   Label
                          | "(" Label { WEAK "," Label } ")" [ "," ] Expression
                        ) .
  ReadStatement     = "READ" "*" { WEAK "," ReadElement } .
  ReadElement       = Designator .
  PrintStatement    = ( "PRINTO" | "PRINT" ) "*" { WEAK "," PrintElement } .
  PrintElement      = stringLit | Expression .
  Expression        = AndExp { ".OR." AndExp } .
  AndExp            = NotExp { ".AND." NotExp } .
  NotExp            = [ ".NOT." ] RelExp .
  RelExp            = AddExp [ RelOp AddExp ] .
  AddExp            = [ AddOp ] MultExp { AddOp MultExp } .
  MultExp           = Factor { MulOp Factor } .
  Factor            = Designator | Constant | "(" Expression ")" .
  Constant          = IntConst | ".TRUE." | ".FALSE." .
  AddOp             = "+" | "-" .
  MulOp             = "*" | "/" .
  RelOp             = ".LT." | ".LE." | ".GT." | ".GE." | ".EQ." | ".NE." .
  Ident             = identifier .
  IntConst          = number .
  Label             = number .
END ToyFortran.
```

Tempting as it might be simply to use Coco/R to produce a program that will parse ToyFortran programs, this week we should like you to produce such a system more directly, by developing a program in the spirit of the one you will find in the textbook in chapter 8.2.

The essence of this program is that it will eventually have a main method that will

- use a command line parameter to retrieve the file name of a data file;
- from this file name derive an output file name with a different extension;
- open these two files;
- initialize the "character handler";
- initialize the "scanner";
- start the "parser" by calling the routine that is to parse the goal symbol;
- close the output file and report that the system parsed correctly (or not, as the case may be).

In this practical you are to develop such a scanner and parser, which you should try in easy stages. So for Task 2, study the grammar above and the skeleton program from the kit (`ToyFortran.cs`) as shown below. In particular, note how the character handler section has been programmed.

```
// Do learn to insert your names and a brief description of what the program is supposed to do!

// This is a skeleton program for developing a parser for ToyFortran programs
// P.D. Terry, Rhodes University, 2017

using Library;
using System;
using System.Text;

class Token {
  public int kind;
  public string val;

  public Token(int kind, string val) {
    this.kind = kind;
    this.val = val;
  } // constructor

} // Token

class ToyFortran {

  // ++++++++++++++++++++++++++ File Handling and Error handlers ++++++++++++++++++++

  static InFile input;
  static OutFile output;

  static string NewFileName(string oldFileName, string ext) {
  // Creates new file name by changing extension of oldFileName to ext
    int i = oldFileName.LastIndexOf('.');
    if (i < 0) return oldFileName + ext; else return oldFileName.Substring(0, i) + ext;
  } // NewFileName

  static void ReportError(string errorMessage) {
  // Displays errorMessage on standard output and on reflected output
    Console.WriteLine(errorMessage);
    output.WriteLine(errorMessage);
  } // ReportError

  static void Abort(string errorMessage) {
  // Abandons parsing after issuing error message
    ReportError(errorMessage);
    output.Close();
    System.Environment.Exit(1);
  } // Abort

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

  const int
    noSym        =  0,
    EOFSym       =  1;
                      // and others like this

  // ++++++++++++++++++++++++++++ Character Handler ++++++++++++++++++++++++++

  const char EOF = '\0';
  const char EOL = '\n';
```

```
    static bool atEndOfFile = false;

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

    static char ch;    // look ahead character for scanner

    static void GetChar() {
    // Obtains next character ch from input, or CHR(0) if EOF reached
    // Reflect ch to output
      if (atEndOfFile) ch = EOF;
      else {
        ch = input.ReadChar();
        atEndOfFile = ch == EOF;
        if (!atEndOfFile) output.Write(ch);
      }
    } // GetChar

    // ++++++++++++++++++++++++++++++++ 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();
      StringBuilder symLex = new StringBuilder();
      int symKind = noSym;

      // ++++++++++++++++++++++++++++++++++++++ over to you!

      sym = new Token(symKind, symLex.ToString());
    } // GetSym

/*  ++++ Commented out for the moment
    // ++++++++++++++++++++++++++++++++ Parser ++++++++++++++++++++++++++++++++++

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

    static void Program() {  }

++++++ */

    // ++++++++++++++++++++ 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");
        System.Environment.Exit(1);
      }
      input = new InFile(args[0]);
      output = new OutFile(NewFileName(args[0], ".out"));

      GetChar();                                  // Lookahead character

//  To test the scanner we can use a loop like the following:
      do {
        GetSym();                                 // Lookahead symbol
        OutFile.StdOut.Write(sym.kind, 3);
        OutFile.StdOut.WriteLine(" " + sym.val);  // See what we got
      } while (sym.kind != EOFSym);

  /*  After the scanner is debugged we shall substitute the following for that loop:

      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");
  */
      output.Close();
    } // Main

  } // ToyFortran
```

Task 3 - First steps towards a scanner

Next, develop the scanner by completing the `GetSym` method, whose goal in life is to recognize tokens. Tokens for this application could be defined by an enumeration of the following, each defined by a unique number:

```
noSym, EOFSym, EOLSym, identSym, numSym, lparenSym, rparenSym, DOSym, IFSym ...
```

As you can see from the grammar given above, Fortran was a "line oriented" language. Unlike the languages you have used previously where the ends of lines are insignificant, you wmust for EOL as a distinctive token.

The scanner can (indeed, must) initially be developed on the pretext that an initial character `ch` has been read. When called, it must (if necessary) read past any "white space" in the input file until it comes to a character that can form part (or all) of a token. It must then read as many characters as are required to identify a token, and assign the corresponding value from the enumeration to the `kind` field of an object called, say, `sym` - and then read the next character `ch` (remember that the parsers we are discussing always look one position ahead in the source).

Test the scanner with a program derived from the skeleton, which should be able to scan the data file and simply tell you what tokens it can find, using the simple loop in the `Main` method as supplied. At this stage do not construct the parser, or attempt to deal with comments. A simple data file for testing can be found in the file `test1.for`, one that simply has a list of all tokens in `test2.for`, and a trickier one in `test3.for` and you can (and probably should) invent a few more for yourself.

You can compile your program by giving the command

```
csharp ToyFortran.cs
```

and can run it by giving a command like

```
ToyFortran test1.for          or     ToyFortran test2.for
```

By the end of the afternoon - Submit an LPRINT listing of your scanner as developed so far

Task 4 - Handling comments and other peculiarities

Next, refine the scanner so that it can deal with (that is, safely ignore) comments in Toy Fortran programs. A suitable data file for testing is to be found in the file `test4.for`, another in `test5.for`.

Comments in Fortran were written with a C in column 1 of a line, as in the examples. Labels on statements (where needed) occurred in columns 1 through 5, and statements themselves were written, one to a line in columns 7 through 72. But allow your system to be free format, other than trying to match the requirement that comments start with a C in column 1 and that the ends of lines mark the ends of statements.

There were some other peculiarities that you might like to deal with. Fortran identifiers could have a maximum of 6 characters, and Fortran string literals were delimited by single quotes - if a single quote were needed within the string this was specified by two single quotes in succession as in 'That''s enough to blow a student''s mind'. Pascal used this convention too - see if you can handle it in your scanner. The file `test6.for` can act as a test case.

You cannot possibly expect to start on Task 5 until such time as the scanner is working properly, so test it thoroughly, please!

Task 5 - At last, a parser

Task 5 is to develop the associated parser as a set of routines, one for each of the non-terminals suggested in the grammar above. These methods should, where necessary, simply call on the `GetSym` scanner routine to deliver the next token from the input. As discussed in chapter 8, the system hinges on the premise that each time a parsing routine is called (including the initial call to the goal routine) there will already be a token waiting in the variable `sym`, and whenever a parsing routine returns, it will have obtained the follower token in readiness for the caller to continue parsing (see discussion on page 103). It is to make communication between all these routines

easy that we declare the lookahead character `ch` and the lookahead token `sym` to be fields "global" to the `ToyFortran` class.

Of course, anyone can write a recognizer for input that is correct. The clever bit is to be able to spot incorrect input, and to react by reporting an appropriate error message. For the purposes of this exercise it will be sufficient to make use of the `accept` routines that you see in the supplied code, ones that simply issue a stern error message, close the output file, and then abandons parsing altogether.

Something to think about: If you have been following the lectures, you will know that associated with each nonterminal *A* is a set *FIRST(A)* of the terminals that can appear first in any string derived from *A*. Alarums and excursions (as they say in the classics). So that's why we learned to use the `IntSet` class in practical 1!

You are reminded of the web pages like

```
http://www.cs.ru.ac.za/courses/CSc301/Translators/sets.htm
http://www.cs.ru.ac.za/courses/CSc301/Translators/inout.htm
```

where you will find useful summaries specifications of library routines especially developed for this course.

To test your parser you might like to make use of the many toy programs supplied in the prac kit. Some of these have deliberate errors, so be careful. Parsing the incorrect ones will be a little more frustrating unless you add syntax error recovery (not required, but feel free to experiment), as the parser will simply stop as soon as it finds the first error. You might like to create a number of "one-liner" data files to make this testing stage easier. Above all, do test your program thoroughly. Any parser for a real compiler that has bugs is almost useless!

```
C:                                                          Sieve2.for
C: Sieve of Eratosthenes for finding primes 2 <= n <= 400 (Toy Fortran version)
C: P.D. Terry,  Rhodes University, 2017
      program sieve2
      logical uncrossed(4000)
C: counters
      integer max, i, n, k, it, iterations, primes
      logical display
      max = 4000
      primes = 0
      print0 *, 'How many iterations?'
      read *, iterations
      display = iterations .eq. 1
      print0 *, 'Supply largest number to be tested '
      read *, n
      if (n .le. max) goto 10
        print *, 'That''s too large, sorry'
        stop
10    do 90 it = 1, iterations
        primes = 0
        print *, 'Prime numbers between 2 and ' , n
        print *, '-----------------------------------'
        i = 2
C: clear sieve
        do 30 i = 2, n
30        uncrossed(i) = .true.
        i = 2
C: the passes over the sieve
50      if (i .gt. n) goto 90
          if (.not. uncrossed(i)) goto 70
            primes = primes + 1
            if (display) then
              print0 *, i
              if (primes .eq. (primes / 8) * 8) print *
            endif
C: now cross out multiples of i
            k = i
60          uncrossed(k) = .false.
            k = k + i
            if (k .le. n) goto 60
70        i = i + 1
        goto 50
90    continue
      print *
      print *, primes, ' primes'
      stop
      end
```