Computer Science 3 - 2015

Programming Language Translation

Practical 5 for week beginning 28 September 2015

Hand in your 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 g13A1234.** 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 analyse a set of C or C++ declaration statements.

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) C and C++ declarations a little better than before.

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 at:

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.

It is generally acknowledged, even by experts, that the syntax of declarations in C and C++ can be quite difficult to understand. This is especially true for programmers who have learned Pascal Modula-2 or C# before turning to a study of C or C++. Simple declarations like

```
int x, list[100];
```

present few difficulties (`x` is a scalar integer, `list` is an array of 100 integers). However, in developing more abstruse (complicated) examples like

```
char **a;      // a is a pointer to a pointer to a character
int *b[10];    // b is an array of 10 pointers to single integers
int (*c)[10];  // c is a pointer to an array of 10 integers
bool *d();     // d is a function returning a pointer to a bool
char (*e)();   // e is a pointer to a function returning a character
```

it is easy to confuse the placement of the various brackets, parentheses and asterisks, perhaps even writing syntactically correct declarations that do not mean what the author intended. By the time one is into writing (or reading) declarations like

```
bool (*(*f())[])();
int  (*(*g[50])())[15];
```

there may be little consolation to be gained from learning that C was designed so that the syntax of declarations (defining occurrences) should mirror the syntax for access to the corresponding quantities in expressions (applied occurrences).

Incidentally, a very readable discussion of how humans can unravel such declarations is to be found in the excellent book by my friend Kim King: "C Programming - a modern approach" (Norton, 1996).

A grammar that describes many of the forms that such declarations can take might be expressed in Cocol as follows:

```
COMPILER Cdecls
/* Describe a subset of the forms that C declarations can assume */

CHARACTERS
  digit  = "0123456789" .
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_" .

IGNORE CHR(0) .. CHR(31)

TOKENS
  number = digit { digit } .
  ident  = letter { letter | digit } .

PRODUCTIONS
  Cdecls   = { DecList } EOF .
  DecList  = Type OneDecl { "," OneDecl } ";"  .
  Type     = "int" | "void" | "bool" | "char" .
  OneDecl  = "*" OneDecl | Direct .
  Direct   = ( ident | "(" OneDecl ")" ) [ Suffix ] .
  Suffix   = Array { Array } | Params .
  Params   = "(" [ OneParam { "," OneParam } ] ")" .
  OneParam = Type [ OneDecl ] .
  Array    = "[" [ number ] "]" .
END Cdecls.
```

Tempting as it might be simply to use Coco/R to produce a program that will analyse declarations, this week we should like you to produce such a recognizer more directly, by developing a program in the spirit of the one you will find in the textbook on pages 92 - 94.

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.

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 (`Declarations.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 C declarations
// P.D. Terry, Rhodes University, 2015

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 Declarations {

  // +++++++++++++++++++++++++ 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';
  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) {
  // 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() {}

++++++ */

  // +++++++++++++++++++++ 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: Declarations 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 this code:

    GetSym();                                   // Lookahead symbol
    CDecls();                                   // Start to parse from the goal symbol
    // if we get back here everything must have been satisfactory
    Console.WriteLine("Parsed correctly");

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

} // Declarations
```

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

```
noSym, intSym, charSym, boolSym, voidSym, numSym, identSym, lparenSym, rparenSym,
lbrackSym, rbrackSym, pointerSym, commaSym, semicolonSym, EOFSym
```

The scanner can (indeed, must) 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 files `SAMPLE0.CPP`, a longer one in `SAMPLE1.CPP`, one that simply has a list of all tokens in `TEST.TXT`.

You can compile your program by giving the command

```
csharp Declarations.cs
```

and can run it by giving a command like

```
Declarations test.txt          or     Declarations sample0.cpp
```

Task 4 - Handling comments

Next, refine the scanner so that it can deal with (that is, safely ignore) Java or C++ style comments (of both sorts) in the set of declarations. Suitable data files for testing are to be found in the files `SAMPLE2.CPP` and `SAMPLE3.CPP`.

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 91). 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 `Declarations` 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 first to develop a simple routine on the lines of the `accept` routine that you see on page 94, that simply issues a stern error message, closes 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!

A note on the IntSet class and other aspects of the library routines

The textbook code extracts are all expressed in C# rather than Java - the two languages are very similar, and I have deliberately stuck to features that are almost identical in both languages). One point of difference comes about in constructing sets. In the C# code in the book you will see code like (page 181):

```
    static SymSet x = new SymSet(eqlSym, neqSym, lssSym, leqSym, gtrSym, geqSym);
```

but at the time, the closest equivalent I was able to incorporate into Java `Library` classes required you to write

```
    static SymSet x= new SymSet(new int[] {eqlSym, neqSym, lssSym, leqSym, gtrSym, geqSym} );
```

However, the advent of Java 5 and 6 afforded the chance to add a "compatible" `IntSet` class to the library, which in the C# version is identical to `SymSet`, but in the Java version you may now write

```
    static IntSet x = new IntSet(eqlSym, neqSym, lssSym, leqSym, gtrSym, geqSym);
```

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.

A note on testing

To test your parser you might like to make use of the data files supplied. One of these (`SAMPLE1.CPP`) has a number of correct declarations. Another (`SAMPLE4.CPP`) has a number of incorrect declarations. Your parser should, of course, be able to parse `SAMPLE1.CPP` easily, and you should be able to "compile" this same file by issuing a command like `BCC SAMPLE1.CPP` just to verify this. Using BCC to "compile" `SAMPLE4.CPP` should be fun, but parsing `SAMPLE4.CPP` with your system will be a little more frustrating unless you added syntax error recovery, 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. Feel free to experiment! But, above all, do test your program out.

Task 6 - Food for thought

Because the course has been disrupted by public holidays, I have decided not to ask you to add syntax error recovery techniques into your parser. The model solution might eventually demonstrate how this could be done, and those of you with time on your hands might like to investigate this for yourselves. Bear in mind that error recovery is fundamental to writing production quality parsers, and that the topic is examinable.

C declarations in general require a rather more complex grammar than the one given above. For example, the grammar will not allow declarations like

```
     int x = 10;                        // initializer
     int x[4] = {1 , 2 , 3 , 4};        // initializer
     static int x[10];                  // modifier
     void func (int **);                // no parameter name
     struct { int a; char b} x, y;      // structures
```

and while it will correctly allow declarations like

```
     void main(int argc, char *argv[]);
```

it will incorrectly allow declarations like

```
     int x[];                           // no size specified
```

You might like to ponder whether the grammar and parser can easily be changed to handle some or more of these situations. Again, I do not expect a solution to be submitted at this stage, but I suggest that you think about it quite hard,