Computer Science 3 - 2013

Programming Language Translation

Practical for Week 23, beginning 30 September 2013

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 g03A1234.** 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 Modula-2 `TYPE` and `VAR` declarations.

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;

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 by following the links at:

`http://www.scifac.ru.ac.za/` or from `http://www.ru.ac.za/`

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 `PRAC23.ZIP` (Java) or `PRAC23C.ZIP` (C#). 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 a subset of the possible ways in which types and variables can be declared in Modula-2:

```
COMPILER Mod2Decl
/* Grammar for a subset of Modula-2 type and variable declarations
   P.D. Terry, Rhodes University, 2013 */

CHARACTERS
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789".

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

COMMENTS FROM '(*' TO '*)'

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Mod2Decl    = { Declaration } .
  Declaration =   "TYPE"  { TypeDecl  SYNC ";" }
                | "VAR"   { VarDecl   SYNC ";" } .
  TypeDecl    = identifier "=" Type .
  VarDecl     = IdentList ":" Type .
  Type        =   SimpleType | ArrayType | RecordType
                | SetType | PointerType .
  SimpleType  = QualIdent [ Subrange ] | Enumeration | Subrange .
  QualIdent   = identifier { "." identifier } .
  Subrange    = "[" Constant ".." Constant "]"  .
  Constant    = number | identifier .
  Enumeration = "(" IdentList ")" .
  IdentList   = identifier { "," identifier } .
  ArrayType   = "ARRAY" SimpleType { "," SimpleType } "OF" Type.
  RecordType  = "RECORD" FieldLists "END" .
  FieldLists  = FieldList { ";" FieldList } .
  FieldList   = [ IdentList ":" Type ] .
  SetType     = "SET" "OF" SimpleType .
  PointerType = "POINTER" "TO" Type .

END Mod2Decl.
```

Tempting as it might be simply to use Coco/R to produce a program that will analyse Modula-2 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 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.

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 (`skeleton.java`) 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 Modula-2 declarations
// P.D. Terry, Rhodes University, 2013

import java.util.*;
import library.*;

class Token {
  public int kind;
  public String val;
```

```
  public Token(int kind, String val) {
    this.kind = kind;
    this.val = val;
  }

} // Token

class MOD2 {

  // ++++++++++++++++++++++++++ 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;
  }

  static void reportError(String errorMessage) {
  // Displays errorMessage on standard output and on reflected output
    System.out.println(errorMessage);
    output.writeLine(errorMessage);
  }

  static void abort(String errorMessage) {
  // Abandons parsing after issuing error message
    reportError(errorMessage);
    output.close();
    System.exit(1);
  }

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

  static final int
    noSym        =  0,
    EOFSym       =  1;

    // and others like this

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

  static final char EOF = '\0';
  static boolean 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);
  }

++++++ */

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

  public static void main(String[] args) {
    // Open input and output files from command line arguments
    if (args.length == 0) {
      System.out.println("Usage: MOD2 FileName");
      System.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

      // report on the properties of the token ++++++++++++  Add something here

    } while (sym.kind != EOFSym);

/*  After the scanner is debugged we shall substitute this code:

    getSym();                                   // Lookahead symbol
    Mod2Decl();                                 // Start to parse from the goal symbol
    // if we get back here everything must have been satisfactory
    System.out.println("Parsed correctly");

*/
    output.close();
  } // main

} // MOD2
```

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, numSym, identSym, ..... 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 `decl1.txt`, a longer one can be found in `decl2.txt`, and you can (and probably should) invent a few more for yourself.

You can compile your program by giving the command

`javac skeleton.java` or `csc skeleton.cs Library.cs`

and can run it by giving a command like

`java MOD2 decl.txt` or `skeleton decl.txt`

Task 4 - handling comments

Next, refine the scanner so that it can deal with (that is, safely ignore) comments in the list of programme offerings. Suitable data files for testing are to be found in the files `decl3.txt` and `decl4.txt`.

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 in Chapter 8.1). It is to make communication between these routines easy that we declare the lookahead character `ch` and the lookahead token `sym` to be fields "global" to the `MOD2` 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 in chapter 8.2, 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 19!

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

The textbook code extracts are all written in C# rather than Java (bet you hadn't really noticed - 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 (section 8.3):

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

but the closest equivalent I was able to incorporate into the original Java `library` classes required you to write

```
static SymSet = 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 now allows you to write

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

The source code in `skeleton.java` is, of course, expressed in Java (the C# version is available in the source kit). Other differences from what you will see in the text come about in trivial things like the use of `boolean` in place of `bool`, `main` in place of `Main`, and so on (a lot of otherwise equivalent methods make different use of uppercase letters in the two languages). 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
```

A note on testing

To test your parser you might like to make use of the data files supplied. One of these (`decl5.txt`) has a number of correct declarations. Another (`decl6.txt`) has a number of incorrect declarations. Your parser should, of course, be able to parse `decl5.txt` easily, but parsing `decl6.txt` 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 so many of you are embroiled in IS projects, 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.

```
(* decl5.txt - some Modula-2 declarations *)

TYPE
  Colours = ( red, orange, yellow, green, blue, indigo, violet );
  FirstColours = Colours [ red .. orange ];

VAR
  Dynamic : POINTER TO ARRAY [ 1 .. 100] OF INTEGER;
  Sieve : SET OF [ 0 .. 4000 ];
  BoolSieve : ARRAY [ 0 .. 4000] OF BOOLEAN;

VAR
  (* It is acceptable to have an empty sequence here *)

TYPE
  SmallInts = [ 1 .. 31 ];
  Persons = RECORD
              Name, Surname : ARRAY [0 .. 10] OF CHAR;
              Age : INTEGER;;
              HairColour : Colours;
            END;

VAR
  I, J, K : Exported.Type;
  Class : ARRAY SmallInts OF Persons;
  Lecturer : Persons;

TYPE
  (* It is acceptable to have an empty sequence here *)
```