Programming Language Translation

Practical 4: week beginning 24th August 2020

All tasks in this practical should be submitted through the RUC submission link by 2 pm on your next practical day.

Objectives

In this practical you will

- improve the Parva language by adding some extensions to the grammar,
- familiarize yourself with the rules and restrictions of LL(1) parsing, and
- help you understand the concept of ambiguity.

Copies of this handout, the Parva language report, the Coco reference manual, and "Pitfalls using Coco" are available on the RUC course page.

Outcomes

When you have completed this practical you should understand:

- how to apply the LL(1) rules manually and automatically;
- how to use Coco/R more effectively.

To hand in (max 30 marks)

This week you are required to hand in via the link on RUConnected:

- Electronic copy of your grammar file (ATG file) for task 1 [16 marks].
- The solutions to tasks 2 [4 marks], 3 [4 marks], and 4 [6 marks] as a pdf document of the edited hand-in sheet available with the prac handout.

You do NOT need to submit listings of any C# code produced by Coco/R. Note that some of the tasks do not need you to use a computer, nor should you. Do them by hand.

As before, not all your tasks will be marked. For this practical, tutors will mark Tasks 2 and 4. I will mark either task 1 or 3 or both.

Feedback for the tasks marked will be provided via RUConnected. 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. A rule not stated there, but which should be obvious, is that you are not allowed to hand in another student's or group'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 on *all* cover sheets and with suitable comments typed into *all* listings.

Note however, that "working with" someone does not imply that you hand in the same solution. The interpretation used in this course (and which determines whether a submission is plagiarised or not) is that you

may discuss how you would approach the problem, but not actually work together on one coded solution. You are still expected to hand in your own solution at all times.

You are expected to be familiar with the University Policy on Plagiarism.

Task 0 Creating a working directory and unpacking the prac kit

There are several files that you need, zipped up this week in the file `PRAC4.ZIP`.

To ensure that all batch scripts provided in the kit execute correctly, copy the zipped prac kit from RUConnected into a new directory/folder in your file space (say `PRAC4`) and unzip the kit there.

Then run all batch scripts provided in the kit from within this new directory. Note these scripts need to be run from the command line – and in order to get used to doing this prior to the exam, I suggest you always use a command line terminal window when doing the practical work.

In the kit for this week, you will find the executable version of Coco/R and batch files for running it, frame files, and various sample programs and grammars, including ones for the grammars given in tasks 1 (Parva), 2 (palindromes) and 4 (reverse polish).

After unpacking this kit attempt to make the parsers, as you did last week. **You can get Coco/R to show you the *FIRST* and *FOLLOW* sets for the non-terminals of the grammar by using the "-options f" flag (change the cmake file to reflect this or just execute coco itself from the command line.** Verify that the objections (if any) that Coco/R raises to these grammars are the same as you have determined by hand.

Task 1 So what if Parva is so restrictive – Fix it! [16 marks]

Parva really is a horrid little language, isn't it? But its simplicity means that it is easy to "extend it".

In the prac kit you will find the grammar for a version of Parva based on the one on page 100 of the notes. Generate a program from this that will recognise or reject Parva programs, and verify that the program behaves correctly with two of the sample programs in the kit, namely VOTER.PAV and VOTER.BAD.

```
cmake Parva
Parva voter.pav
Parva voter.bad -L
```

Now modify the grammar to add various features. Specifically, add (and check that each addition works):

- All those fun arithmetic assignment operators: +=, -= *= and /=.
- A restriction that an identifier may contain underscore characters, but may not end with one, so that My_Name would be acceptable, but not My___name_.
- Optional *elsif* and *else* clauses for the *if* statement.
- A *do-while* loop, and *break* and *continue* statements.
- A *for* loop inspired by the one in Pascal (look it up!).

Here are two *silly* examples of code that should give you some ideas:

```
void Main() {
// Demonstrate various statements
  const too_Much = 55;
  int mark__1 = 5, mark__2 = 10, mark__3 = 20;
```

```
  mark__1, mark__2, mark__3 = mark__2, mark__1 + mark__3, 256;
  age = 0;
  while (age < 180) {
    age = age + 1;
    write("Happy birthday ");
    if (age == 75) {
      write("You must have found the secret of longevity!");
      break;
    }
    write("Have another good year\n");
  }
} // Main


void Main () {
// (not supposed to do anything useful!)
  int age;
  bool haveVoted;
  read("How old are you, and have you ever voted? ", age, haveVoted);
  if (age == 18) {
    write("Old enough to vote");
    if (!haveVoted) write(" but not had the chance!");
  }
  elsif (age == 21) {
    write("party time!");
    int headache = 0, strain = 0;
    for beers = 20 downto 0 {
      strain += 1; headache += 2;
     }
   }
  elsif ((age > 21) && (age < 40))
    write("time to start work");
  elsif (age > 70)
    write("you deserve a rest");
  else
    write("life must be boring");
} // Main
```

These little programs and some other like them are in the kit, and you can easily write some more of your own. Actually, the ones above are rather ambitious. Start on something really simple, like:

```
void Main () {
  if (a)
    c = d;
  else c = 12;
} // Main
```

and

```
void Main () {
```

```
    int i = 0;
    do
      i *= 2;
    while (i < 10);
  } // Main
```

Note: Read that phrase again: "that should give you some ideas". And again. And again. Don't just rush in and write a grammar that will recognise only some restricted forms of statement. Think hard about what sorts of things you can see there, and think hard about how you could make your grammar fairly general.

Hint: All we require at this stage is the ability to describe these features. You do not have to try to give them any semantic meaning or write code to allow you to use them in any way. In later pracs we might try to do that, but please stick to what is asked for this time, and don't go being over ambitious.

Warning. Language design and grammar design is easy to get wrong. Think hard about these problems.

Task 2 Palindromes [4 marks]

Palindromes are character strings that read the same from either end, like "Hannah" or “madam". The following represent various attempts to find grammars (see Palinx.atg) that describe palindromes made only of the letters a and b:

```
(1)  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" .
(2)  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" | "a" | "b" .
(3)  Palindrome = "a" [ Palindrome ] "a" | "b" [ Palindrome ] "b" .
(4)  Palindrome = [ "a" Palindrome "a" | "b" Palindrome "b" | "a" | "b" ] .
```

Which grammars achieve their aim? If they do not, explain why not. Which of them are LL(1)? Can you find other (perhaps better) grammars that describe palindromes and which are LL(1)?

Task 3 Thinking about ambiguity [4 marks]

Which of the following statements are true? Justify your answer.

(a) An LL(1) grammar cannot be ambiguous.
(b) A non-LL(1) grammar must be ambiguous.
(c) An ambiguous language cannot be described by an LL(1) grammar.
(d) It is possible to find an LL(1) grammar to describe any non-ambiguous language.

Task 4 Reverse Polish Notation [6 marks]

You may be familiar with RPN or "Reverse Polish Notation" as a notation that can describe expressions without the need for parentheses. The notation eliminates parentheses by using "postfix" operators after the operands. To evaluate such expressions one uses a stack architecture, such as that which forms the basis of the PVM machine studied in the course. Examples of RPN expressions are:

```
3 4 +        equivalent to    3 + 4
```

```
3 4 5 + *    equivalent to    3 * (4 + 5)
```

In many cases an operator is taken to be "binary" - applied to the two preceding operands - but the notation is sometimes extended to incorporate "unary" operators - applied to one preceding operand:

```
4 sqrt       equivalent to    sqrt(4)
5 -          equivalent to    -5
```

Here are two attempts to write grammars (see RPNx.atg) describing an RPN expression:

```
(G1)     RPN       =     RPN RPN binOp
                       | RPN unaryOp
                       | number .
         binOp     =   "+" | "-" | "*" | "/" .
         unaryOp   =   "-" | "sqrt" .
```

and

```
(G2)     RPN       =   number REST .
         REST      =   [ number REST binOp REST | unaryOp ].
         binOp     =   "+" | "-" | "*" | "/" .
         unaryOp   =   "-" | "sqrt" .
```

Are these grammars equivalent? Is either (or both) ambiguous? Do either or both conform to the LL(1) conditions? If not, explain clearly where the rules are broken, and come up with an LL(1) grammar that describes RPN notation, or else explain why it might be necessary to modify the language itself to overcome any problems you have uncovered.
