Computer Science 3 - 2016

Programming Language Translation

Practical for Week 4, beginning 8 August 2016

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

- familiarize you with the rules and restrictions of LL(1) parsing, and
- help you understand the concept of ambiguity, and
- get more experience in writing simple grammars.

You will need this prac sheet and your text book. As usual, copies of the prac sheet are also available at `http://www.cs.ru.ac.za/courses/CSc301/Translators/trans.htm`.

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:

This week you are required to hand in, besides the cover sheet:

- Listings of your Cocol grammars, produced on the laser printer by using the LPRINT utility. Some of these listings will get quite "wide" so please set them out nicely.
- Electronic copies of your grammar files (ATG files), stored in a folder under one of the group's student numbers.
- Discussions of the questions raised in Tasks 4 to 8, which you should incorporate into the file `PRAC4.TXT`.

I do NOT require listings of any C# code produced by Coco/R.

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 University web site.

Task 1 - Grab a mug of hot Coco and press on

Copy the prac kit `PRAC4.ZIP` into a newly created directory/folder in your file space in the usual way, for example:

```
j:
md  prac4
cd  prac4
copy  i:\csc301\trans\prac4.zip
unzip  prac4.zip
```

You will find the executable version of Coco/R and batch files for running it, frame files, etc.

Task 2 - Describe BNF

In last week's practical you met a Cocol grammar like the following. This describes a set of productions written in Wirth's EBNF notation - and which itself makes use of the Wirth meta-brackets **{ }** and **[ ]** .

```
COMPILER EBNF $CN
/* Recognize a set of EBNF productions
   P.D. Terry, Rhodes University, 2016 */

CHARACTERS
  control  = CHR(0) .. CHR(31) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  digit    = "0123456789" .
  noQuote1 = ANY - "'" - control .
  noQuote2 = ANY - '"' - control .

TOKENS
  nonTerminal = letter { letter | lowline | digit } .
  terminal    = "'" noQuote1 { noQuote1 } "'" | '"' noQuote2 { noQuote2 } '"' .

COMMENTS FROM "(*" TO "*)"  NESTED

IGNORE control

PRODUCTIONS
  EBNF       = { Production } EOF .
  Production = nonTerminal "=" Expression SYNC "." .
  Expression = Term { "|" Term } .
  Term       = Factor { Factor } .
  Factor     =   nonTerminal
               | terminal
               | "[" Expression "]"
               | "(" Expression ")"
               | "{" Expression "}" .
END EBNF.
```

Use this as a guide to develop a new Cocol description, this time of a system that will recognize or reject a set of productions written one to a line in "traditional" BNF notation, like those below:

```
   <name>               ::= <title> <first part> <surname>
   <title>              ::= Mr | Miss | Ms | Dr | Prof | eps
   <first part>         ::= <name> | <first part> <name> | eps
   <surname>            ::= <name>
   <name>               ::= <letter> | <letter> <name>
   <letter>             ::= <first half letter> | <second half letter>
   <first half letter > ::= a | b | c | d | e | f | g | h | i | j | k | l | m
   <second half letter> ::= n | o | p | q | r | s | t | u | v | w | x | y | z
```

Examples like this can be found in the files `BNF.txt` and `BNF.bad` *etc*. Note that we have used `eps` to denote the Greek letter ε (*epsilon*) which cannot easily be represented in the character set you have available.

Your grammar may make use of the Wirth meta-brackets; the productions it describes cannot incorporate them.

Task 3 - Let's take a trip to the C

The Parva grammar you met last week incorporated the following productions to describe the syntax of an *Expression*. This uses the hierarchy of operator precedence that is used in Wirth's languages Pascal, Modula-2 and Oberon.

```
Expression        = AddExp [ RelOp AddExp ] .
AddExp            = [ "+" | "-" ] Term { AddOp Term } .
Term              = Factor { MulOp Factor } .
Factor            =   Designator | Constant
                    | "new" BasicType "[" Expression "]"
                    | "!" Factor | "(" Expression ")" .
AddOp             = "+" | "-" | "||" .
MulOp             = "*" | "/" | "&&" | "%" .
RelOp             = "==" | "!=" | "<" | "<=" | ">" | ">=" .
```

Without adding any more operators - that is, restricting yourself to `+ - * / % && || !` and the relational operators `== != < > <= and >=` - develop the part of a Parva grammar for *Expression* that would recognize *Expressions* where the operator precedence is the same as it is in C#. You may need to look up the operator precedence rules in C#.

If you wish, incorporate this grammar for *Expression* into the Parva grammar you had last week for test purposes.

From a purely syntactical viewpoint, are the two grammars equivalent - can you think of an expression that one grammar would recognize but the other would reject?

Task 4 - Expressions - again

The next few tasks ask you to analyse various grammars by hand. Some of them have distinct problems. Besides trying to detect those problems "by hand", learn to use Coco to help debug them. Ask the demonstrators to show you how to use Coco in test mode, and also to determine the *FIRST* and *FOLLOW* sets for the non-terminals of the grammar, and verify that the objections (if any) that Coco/R raises to these grammars are the same as you have determined by hand.

The following grammar attempts to describe expressions incorporating the familiar operators with their correct precedence and associativity, as well as an exponentiation operator, allowing, for examples `a + b↑c`.

```
COMPILER Expression
IGNORE CHR(0) .. CHR(31)
PRODUCTIONS
  Expression = Term    { ( "+" | "-" ) Term  } .
  Term       = Factor  { ( "*" | "/" ) Factor } .
  Factor     = Primary [ "↑" Expression ] .
  Primary    = "a" | "b" | "c" .
END Expression.
```

Is this an ambiguous grammar? (Hint: try to find an expression that can be parsed in more than one way). Is it an LL(1) grammar? If not, why not, and can you find a suitable grammar that *is* LL(1)?

Task 5 - Meet the family

Consider the following grammar:

```
COMPILER Home
IGNORE CHR(0) .. CHR(31)
PRODUCTIONS
  Home      = Family { Pets } [ Vehicle ] "house" .
  Pets      = "dog" [ "cat" ] | "cat" .
  Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
  Family    = Parents { Children } .
  Parents   = [ "Dad" ] [ "Mom" ] | "Mom" "Dad" .
  Child     =   "Helen" | "Margaret" | "Alice" | "Robyn" | "Cathy"
              | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
END Home.
```

Analyse this grammar in detail. Is it LL(1) compliant? If not, why not?

Task 6 - Palindromes

Palindromes are character strings that read the same from either end, like "Hannah" or my brother Peter's favourite line when he did the CTM advertisements: "Bob Bob". The following represent various attempts to find grammars 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 better describe palindromes and which *are* LL(1)?

Task 7 - Reverse Polish Notation

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 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 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.

Task 8 - Pause for thought

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.

Questions - Available as an ASCII file PRAC4.TXT or Word file PRAC4.DOC to edit

Task 3 - Let's take a trip to the C

From a purely syntactical viewpoint, are the C and Pascal expressions equivalent - can you think of an expression that one grammar would recognize but the other would reject?

Task 4 - Expressions - again

Is this an ambiguous grammar? If so, why? is it an LL(1) grammar? If not, why not, and can you find a suitable grammar that *is* LL(1)?

Task 5 - Meet the family

What form does your grammar take when you eliminate the meta-brackets?

Which, if any, productions break the LL(1) rules, and why?

Can you find an equivalent grammar that does obey the LL(1) constraints? If so, give it. If not, explain why you think it canot be done.

Task 6 - Palindromes

Does grammar 1 describe palindromes? If not, why not?

Is it an LL(1) grammar? If not, why not?

Does grammar 2 describe palindromes? If not, why not?

Is it an LL(1) grammar? If not, why not?

Does grammar 3 describe palindromes? If not, why not?

Is it an LL(1) grammar? If not, why not?

Does grammar 4 describe palindromes? If not, why not?

Is it an LL(1) grammar? If not, why not?

Can you find a better grammar to describe palindromes? If so, give it, if not, explain why not.

Task 7

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.

Task 8

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

(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.