Computer Science 3 - 2009

Programming Language Translation

Practical for Week 22, beginning 28 September 2009 - Solutions

This tutorial/practical was not always well done. Many people could "guess" the answers, but could not or did not justify their conclusions. **If set in exams, in these sorts of questions it is important to do so.**

As usual, you can find a "solution kit" as `PRAC22A.ZIP` or `PRAC22AC.ZIP` if you wish to experiment further.

This might be an appropriate point to mention that Coco/R allows two pragma directives. One of these, $T will perform grammar tests but suppress the generation of the parser and scanner, and the other $F will produce a list of FIRST and FOLLOW sets for the non-terminals of your grammar - very useful when debugging LL(1) problems.

Task 1 Steam Radio

The original grammar had productions

```
Radio        = { TalkShow | NewsBulletin | "music" | "advert" } EOF .
NewsBulletin = "advert" NewsItem { NewsItem } [ Weather ] Filler .
NewsItem     = "ANC" [ "cosatu" ] | "strike" | [ "cosatu" ] "ANC" | "zuma" | "corruption" | "murder" .
TalkShow     = "host" { "listener" "host" } [ Filler ] .
Filler       = "music" | "advert"  .
Weather      = { "snow" | "rain" | "cloudy" | "windy" } .
```

This is pretty obviously non-LL(1) and most people could see one of the obvious reasons, and many seem to have looked no further. But here is what I hoped you might have done.

If we rewrite the grammar without meta-brackets we get something like:

```
Radio        = Programmes EOF .
Programmes   = Programme Programmes | ε .
Programme    = TalkShow | NewsBulletin | "music" | "advert" .
NewsBulletin = "advert" NewsItems OptWeather Filler .
NewsItems    = NewsItem NewsItems | ε .
OptWeather   = Weather | ε .
NewsItem     = "ANC" OptCosatu | "strike" | OptCosatu "ANC" | "zuma" | "corruption" | "murder" .
OptCosatu    = "cosatu" | ε .
TalkShow     = "host" Exchanges OptFiller .
Exchanges    = Exchange Exchanges | ε .
OptFiller    = Filler | ε .
Exchange     = "listener" "host" .
Filler       = "music" | "advert"  .
Weather      = OneWeather Weather | ε .
OneWeather   = "snow" | "rain" | "cloudy" | "windy" .
```

Rule 1 is broken in only two places:

(a) One of the options for *Programme*, that is *NewsBulletin*, starts with "advert", which is an option in its own right for *Programme*.

(b) Since *OptCosatu* is nullable, two of the options in *NewsItem* can both start with "ANC"

To check Rule 2 we need to look at the nullable non-terminals, which are

Programmes, NewsItems, OptWeather, Weather, OptCosatu, Exchanges, OptFiller

You should check through in detail, but the two that cause problems are *OptCosatu* and *OptFiller*:

```
FIRST(OptCosatu)  = { cosatu }
FOLLOW(OptCosatu) = { music, advert, ANC, strike, zuma, corruption, murder, cosatu,
                      snow, rain, cloudy, windy }

FIRST(OptFiller)  = { music, advert }
FOLLOW(OptFiller) = { music, advert, host }
```

As mentioned in class, with a bit of practice one can see many of these problems just by looking at the EBNF version of the productions. A look at the three productions

```
Radio       = { TalkShow | NewsBulletin | "music" | "advert" } EOF .
NewsItem    = "ANC" [ "cosatu" ] | "strike" | [ "cosatu" ] "ANC" | "zuma" | "corruption" | "murder" .
NewsBulletin = "advert" NewsItem { NewsItem } [ Weather ] Filler .
```

quickly shows up the Rule 1 problems. The others may not be so obvious, although the second of these last three productions shows that *["cosatu"]* is nullable, and one quickly sicovers that Rule 2 is broken. Similarly, a look at

```
TalkShow = "host" { "listener" "host" } [ Filler ] .
```

shows that *[Filler]* is nullable, and this combined with the production for *Radio* shows that Rule 2 must be broken for this nullable component as well.

How, if at all, can one find a better grammar? Several submissions simply gave up at this point, but their authors often gave very spurious (that is, indefensible) reasons. Just because a grammar is not LL(1) is no guarantee that you cannot find an equivalent LL(1) grammar - but by the same token, it is possible to find many non-LL(1) grammars that can be converted to LL(1) grammars quite easily (go and read section 7.3 of the textbook again), although in some cases this is not possible (read the discussion about the "dangling else" again).

Some of the problems in the grammar are easily overcome. If we change the production for *TalkShow* to

```
TalkShow = "host" { "listener" "host" } .
```

we remove one of the Rule 2 problems, and we do not lose anything at all - if a `Filler` had been present it would have shown up as part of the next item.

We can remove the Rule 1 problem involving "advert" by a simple trick which is useful on many occasions - delay the factorization of the alternatives that cause problems

```
Radio      = { TalkShow | "music" | "advert" [ RestOfNews ] } EOF .
RestofNews = NewsItem { NewsItem } [ Weather ] Filler .
```

Conceptually a *NewsBulletin* in the old form - starting with an "advert" - is still required.

How do we get the problem of "cosatu" to go away? A careful look at the production for *NewsItem* reveals that the grammar is actually ambiguous - it appears we are allowed to parse a substring "ANC" "cosatu" "ANC" in two ways (either as ("ANC" "cosatu") (ε "ANC") or as ("ANC" ε) ("cosatu" "ANC"). Now if we really want to find a grammar that is ambiguous (bearing in mind the rude remarks made in lectures that politicians love ambiguity, we might just want to do so!) we shall not be able to do this in an LL(1) compliant way. Trying to get behind the intent of the problem might suggest that we want a grammar in which it is possible to have news items on "ANC" without mentioning "cosatu" in the same breath, but if we ever mention "cosatu" it must either have been just after mentioning "ANC", or (if we had not done so), we have no option but to mention "ANC" straight afterwards.

One way to do this is to write the production:

```
NewsItem  = "ANC" [ "cosatu" ] | "strike" | "cosatu" "ANC" | "zuma" | "corruption" | "murder" .
```

This gets rid of the Rule 1 violation, but not the Rule 2 violation. However, this one might not be serious - it is, in fact, exactly the same situation as the "dangling else" and a practical parser would resolve it in the same way that we discuss in lectures - that is, a news item on "cosatu" immediately following "ANC" would be bound to that story on "ANC", and not to one still to follow. So, presented with

"ANC" "cosatu" "ANC" "zuma"

would be handled as syntactically equivalent to

("ANC" "cosatu") ("ANC") ("zuma")

and not as

("ANC") ("cosatu" "ANC") "zuma"

Of course, in a sense, this behaviour of the parser might not be what was really wanted - this is a situation where the semantics have become "context sensitive". Int he cas of the "dangling else" the recursive descent parser works very well, and if one wants the opposite sort of effect one can resort to code in { braces }.

Task 2 - Palindromes

Palindromes are character strings that read the same from either end. You were invited to explore various ways of finding 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 describe palindromes and which *are* LL(1)?

This is one of those awful problems that looks deceptively simple, and indeed is deceptive. We need to be able to cater for palindromes of odd or even length, and we need to be able to cater for palindromes of finite length, so that the "repetition" that one immediately thinks of has to be able to terminate.

Here are some that don't work:

```
COMPILER Palindrome /* does not terminate */
PRODUCTIONS
  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" .
END Palindrome.


COMPILER Palindrome /* only allows odd length palindromes */
PRODUCTIONS
  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" | "a" | "b" .
END Palindrome.

COMPILER Palindrome /* only allows even length palindromes */
PRODUCTIONS
  Palindrome = "a" [ Palindrome ] "a" | "b" [ Palindrome ] "b" .
END Palindrome.
```

Of those grammars, the first seems to obey the LL(1) rules, but it is useless (it is not "reduced" in the sense of the definitions on page 129). The second one is obviously non-LL(1) as the terminals "a" and "b" can start more than one alternative. The third one is less obviously non-LL(1). If you rewrite it

```
COMPILER Palindrome /* only allows even length palindromes */
PRODUCTIONS
  Palindrome = "a" Extra "a" | "b" Extra "b" .
  Extra      = Palindrome | ε .
END Palindrome.
```

and note that `Extra` is nullable, then `FIRST(Extra) = { "a", "b" }` and `FOLLOW(Extra) = { "a", "b" }`.

Here is another attempt

```
COMPILER Palindrome /* allows any length palindromes */
PRODUCTIONS
  Palindrome = [ "a"  Palindrome  "a" | "b"  Palindrome  "b" | "a" | "b" ] .
END Palindrome.
```

This describes both odd and even length palindromes, but is non-LL(1). `Palindrome` is nullable, and both `FIRST(Palindrome)` and `FOLLOW(Palindrome) = { "a", "b" }`. And, as most were quick to notice, it breaks Rule 1 immediately as well.

Other suggestions were:

```
COMPILER Palindrome /* allows any length palindromes */
PRODUCTIONS
  Palindrome =  "a"  [ Palindrome  "a"] | "b"  [ Palindrome  "b" ] .
END Palindrome.
```

but, ingenious as this appears, it does not work either. Rewritten it would become

```
COMPILER Palindrome /* allows any length palindromes */
PRODUCTIONS
  Palindrome =  "a"  PalA | "b" PalB .
  PalA       = Palindrome  "a" | .
  PalB       = Palindrome  "b" | .
END Palindrome.
```

`PalA` and `PalB` are both nullable, and `FIRST(PalA) = { "a" , "b" }` while `FOLLOW(PalA) = FOLLOW(Palindrome) = { "a", "b" }` as well.

In fact, when you think about it, you simply will not be able to find an LL(1) grammar for this language. (That is fine; grammars don't have to be LL(1) to be valid grammars. They just have to be LL(1) or very close to LL(1) to be able to write recursive descent parsers.) Here's how to think about it. Suppose I asked you to hold your breath for as long as you could, and also to nod your head when you were half way through. I don't believe you could do it - you don't know before you begin exactly how long you will be holding your breath. Similarly, if I told you to get into my car and drive it till the tank was empty but to hoot the hooter when you were half way to running out you could not do it. Or if I told you to walk into a forest with your partner and kiss him/her when you were in the dead centre of the forest, you would not know when the magic moment had arrived.

LL(1) parsers have to be able to decide just by looking at one token exactly what to do next - if they have to guess when they are are half-way through parsing some structure they will not be able to do so. One would have to stop applying the options like `Palindrome = "a" Palindrome "a"` at the point where one had generated or analyzed half the palindrome, and if there is no distinctive character in the middle one would not expect the parser to be able to do so.

If course, if one changes the problem ever so slightly in that way one *can* find an LL(1) grammar. Suppose we want a grammar for palindromes that have matching *a* and *b* characters on either end and a distinctive *c* or pair of *c* characters in the centre:

```
COMPILER Palindrome /* allows any length palindromes, but c must be in the middle */
PRODUCTIONS
  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" | "c" [ "c" ] .
END Palindrome.
```

Task 3 - 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.

To answer this sort of question you must be able to argue convincingly, and most people did not do that at all!

(a) is TRUE. An LL(1) grammar cannot be ambiguous. If a language can be described by an LL(1) grammar it will always be able to find a single valid parse tree for any valid sentence, and no parse tree for an invalid sentence. The rules imply that no indecision can exist at any stage - either you can find a unique way to continue the implicit derivation from the goal symbol, or you have to conclude that the sentence is malformed.

But you cannot immediately conclude any of the "opposite" statements, other than (c) which is TRUE. If you *really* want to define an ambiguous language (and you may have perfectly good/nefarious reasons for doing so - stand-up comedians do it all the time) you will not be able to describe it by an LL(1) grammar, which has the property that it can only be used for deterministic parsing.

In particular (b) is FALSE. We can "justify" this by giving just a single counter example to the claim that it might be true. We have seen several such grammars. The palindrome grammars above are like this - even though

they are non LL(1) for the reasons given, they are quite unambiguous - you would only be able to parse any palindrome in one way! Many people seem not to realise this - they incorrectly conclude that non-LL(1) inevitably implies ambiguity. The other classic case is that of the left-recursive expression grammars discussed in class and in chapter 6.1.

Similarly (d) is FALSE. Once again the palindrome example suffices - this language is simple, unambiguous, but we can easily argue that it is impossible to find an LL(1) grammar to describe it.

Task 5 - Phools rush in (where angels fear to tread, apparently)

```
COMPILER Phoolish $CN
CHARACTERS
  letter    = "abcdefghijklmnopqrstuvwxyz" .
  consonant = "BCDFGHJKLMNPQRSTVWXYZ" .
  vowel     = "AEIOU" .
TOKENS
  Lletter    = letter .
  Uconsonant = consonant .
  Uvowel     = vowel .
IGNORE CHR(0) .. CHR(31)
PRODUCTIONS
  Phoolish = { Sentence } "." .
  Sentence = Lletter | Uvowel Sentence | Uconsonant Sentence Sentence .
END Phoolish.
```

Many people had misread the question. A complete sentence can follow an uppercase vowel, and two sentences can follow an uppercase consonant.

Thus, for example, while *a Ab Cde* are valid simpler sentences, so too are *U Ab* (Vowel Sentence) and *W UAb Cde* (Consonant Sentence Sentence). All very phoolish, you will agree!

Notice that the names of character sets cannot appear in the PRODUCTIONS section, seductive though this may appear. So the following is incorrect:

```
COMPILER Phoolish $CN
CHARACTERS
  letter    = "abcdefghijklmnopqrstuvwxyz" .
  consonant = "BCDFGHJKLMNPQRSTVWXYZ" .
  vowel     = "AEIOU" .
IGNORE CHR(0) .. CHR(31)
PRODUCTIONS  /* INCORRECT !!!!!!!! */
  Phoolish = { Sentence } "." .
  Sentence = letter | vowel Sentence | consonant Sentence Sentence .
END Phoolish.
```

Task 6 - How are things stacking up?

The grammar for the assembler language was well done by some, and rather inadequately done by others. In particular, few submissions had handled the PRNS opcode correctly - this one takes a "string" as an argument, and the best way to define a string is in the TOKENS section (and, in fact, a way of doing this had been given to you in the Parva grammar last week).

It is best to split the opcodes into 4 groups - those that take no argument, those that take only a numerical argument, those that take a numerical argument or a label, and PRNS. Few submissions did this, which wa disappointing.

There are two complications that were not handled all that well - one should treat the end-of-line as significant, and one should be able to handle completely blank lines. Here is one solution, which fudges it somewhat, by not attaching an eol to each statement but almost regarding an eol as a statement in its own right. This also means that one can have labels on a line by themselves:

```
COMPILER Assem $NC
/* Simple assembler for the PVM
   P.D. Terry, Rhodes University, 2004 */

CHARACTERS
  lf       = CHR(10) .
  control  = CHR(0) .. CHR(31) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
```

```
  digit    = "0123456789" .
  stringCh = ANY - '"' - control .
TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh } '"' .
  eol        = lf .
COMMENTS FROM ";" TO lf
IGNORE CHR(9) + CHR(11) .. CHR(13)

PRODUCTIONS
  Assem        = [ "ASSEM" eol ] [ "BEGIN" eol ] { Statement | eol } [ "END" "." eol ] .
  Statement    = MemAddress ( OneWord | TwoWord | WriteString | Label | Branch ) .
  MemAddress   = [ "{" number "}" | number ] .

  OneWord      =   "ADD"  | "AND"  | "ANEW" | "CEQ"  | "CGE"  | "CGT"  | "CLE"
                 | "CLT"  | "CNE"  | "DIV"  | "HALT" | "INPB" | "INPI" | "LDV"
                 | "LDXA" | "MUL"  | "NEG"  | "NOP"  | "NOT"  | "OR"   | "PRNB"
                 | "PRNI" | "PRNL" | "REM"  | "STO"  | "SUB" .
  TwoWord      = ( "DSP" | "LDC" | "LDA" | "LDL" | "STL" ) SignedNumber .
  SignedNumber = [ "+" | "-" ] number .
  WriteString  = "PRNS" stringLit .
  Label        = identifier .
  Branch       = ( "BRN" | "BZE" ) ( number | identifier ) .
END Assem.
```

Here is another method that may appeal more, where the `eol` is explicitly attached to a statement:

```
  Assem1       = [ "ASSEM" eol ] [ "BEGIN" eol ] { Statement } [ "END" "." eol ] .
  Statement    = [ MemAddress ] [ Label ] [ OneWord | TwoWord | WriteString | Branch ] eol .
  MemAddress   = "{" number "}" | number .
```

Some submissions attempted to distinguish between the .COD input format and the .PVM input format. The solutions above do not do this, and are quite permissive. Here is one way of making the distinction:

```
  Assem2       = CODFormat | PVMFormat .
  CODFormat    = "ASSEM" eol "BEGIN" eol PVMFormat "END" "." eol .
  PVMFormat    = { Statement } .
  Statement    = [ MemAddress ] [ Label ] [ OneWord | TwoWord | WriteString | Branch ] eol .
  MemAddress   = "{" number "}" | number .
```

but even this may strike you as too permissive; perhaps one should try something more like:

```
  Assem3       = CODFormat | PVMFormat .
  CODFormat    = "ASSEM" eol "BEGIN" eol { CODStatement } "END" "." eol .
  PVMFormat    = { PVMStatement } .
  CODStatement = [ "{" number "}" ] Statement .
  PVMStatement = [ number ] Statement .
  Statement    = [ Label ] [ OneWord | TwoWord | WriteString | Branch ] eol .
```

Finally, note the way in which I have used the COMMENTS facility to handle one form of comment, and `MemAddress` to handle the ignorable "comment" at the start of a line. The COMMENTS facility allows comments to appear "anywhere". If we used COMMENTS to describe the `MemAddress` idea we would be able to have statements like

```
LDA { 3 } 23
```

which is not desirable. One might, however, introduce a `comment` token

```
CHARACTERS
  inComment = ANY - control .
TOKENS
  comment    = ";" { inComment } .
PRODUCTIONS
  Statement    = MemAddress ( OneWord | TwoWord | WriteString | Label | Branch ) [ comment ] .
```

since comments from ; to end of line can only appear in one place in each line.

Finally, note that while it is usual to find `HALT` as the last statement in a program, this is not demanded!

Task 8 - Reverse Polish Notation

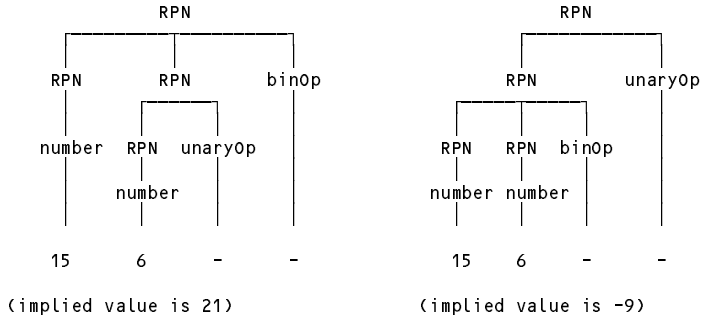
The grammars are not equivalent. To show this we need only find one string that one will accept but the other will not, as many people realised. An example of such a string would be `45 sqrt sqrt`, which can only be recognized by G1 in one way, but not at all by G2.

Using the expression

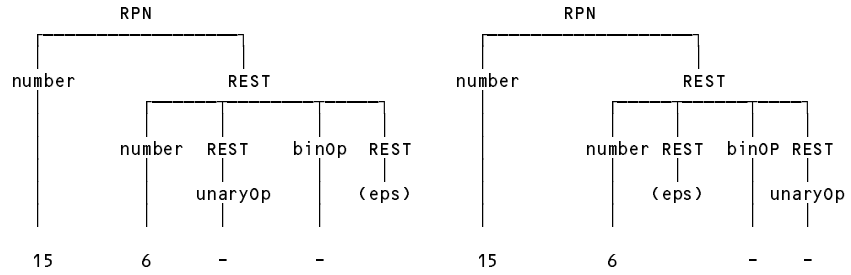
```
15  6  -  -
```

as an example, and by drawing appropriate parse trees, we can demonstrate that both of the grammars are ambiguous.

Using grammar G1:



Using grammar G2:



To analyse each of these grammars to check whether they conform to the LL(1) conditions:

G1 is left recursive and thus is immediately ruled out as a possible LL(1) grammar. There are two alternatives for the right side of the production for RPN that both start with RPN, so Rule 1 is broken, reggardless of the exact nature of FIRST(RPN) - which in any case is `number`, and all three alternatives start with this

For G2, REST is nullable. First(REST) is { number, sqrt, - } while Follow(REST) is { +, -, / , * } so Rule 2 is broken.

To try to overcome the problem we seem at the outset to try to to find a different token to represent the operation of unary negation, asn several people realised. If, for example, we use `negate` for the "unary minus", then G1 becomes non-ambiguous (but still not LL(1)) but G2, while now being LL(1), will still not recognize `4 sqrt sqrt`. Like the palindrome example, this one is frustrating - it looks so easy. Again we emphasize that a valid grammar does not *have* to be LL(1) - it just helps when building recursive descent parsers.

I have not yet found an LL(1) grammar for this language, though I have a sneaky feeling I just have not tried hard enough!

Any contributions gratefully received.

Task 9 - Eliminating Metabrackets

Transforming PRODUCTIONS with meta brackets is really easy - there is a simple fail-safe method for doing it (see page 156). Applied to

```
PRODUCTIONS
  Calc1      = { Expression "=" } EOF .
  Expression = ["+" | "-" ] Term { "+" Term  | "-" Term } .
  Term       = Factor { "*" Factor | "/" Factor } .
  Factor     = Primary { "!" } .
  Primary    = decNumber | hexNumber | [ "abs" ] "(" Expression ")" .
END Calc1.
```

we get something like

```
PRODUCTIONS
  Calc2       = Expressions EOF .
  Expressions = Expression "=" Expressions | .
  Expression  = Sign Term MoreTerms .
  MoreTerms   = ( "+" Term  | "-" Term ) MoreTerms | .
  Sign        = "+" | "-" | .
  Term        = Factor MoreFactors .
  MoreFactors = ( "*" Factor | "/" Factor ) MoreFactors | .
  Factor      = Primary Bangs .
  Primary     = decNumber | hexNumber | Function "(" Expression ")" .
  Bangs       = "!" Bangs | .
  Function    = "abs" | .
END Calc2.
```

Of course this grammar is now right associative, which is undesirable for the purposes of tree building.

I apologize - some people thought they had to eliminate the {} used in the TOKENS section, but that section defines tokens in the form of regular expressions, and these don't have to obey LL(1) constraints.