Computer Science 3 - 2017

Programming Language Translation

Practical for Week 4, beginning 7 August 2017 - Solutions

This practical was not always well done. Many people could "guess" the answers, but could not or did not justify their conclusions. **If these sorts of questions are set in exams it is important to justify your claims, so as not to be seen to be guessing.**

As usual, you can find a "solution kit" as `PRAC4A.ZIP` if you wish to experiment further.

Task 1 - Self-describing EBNF

You were given a Cocol grammar that describes (in EBNF) a set of productions written in EBNF. Part of this read:

```
TOKENS
  nonterminal = letter { letter | lowline | digit } .
  terminal    = '"' noquote { noquote } '"' .

PRODUCTIONS
   EBNF0      = { Production } EOF .
   Production = nonterminal "=" Expression "." .
   Expression = Term { "|" Term } .
   Term       = Factor { Factor } .
   Factor     =   nonterminal
                | terminal
                | "[" Expression "]"
                | "(" Expression ")"
                | "{" Expression "}" .
END EBNF0.
```

(a) Do the production in the EBNF grammar obey the LL(1) constraints? Justify your answer.

Yes, they do. It will suffice to argue as follows: The only productions we need consider for Rule 1 are the 5 alternatives for *Factor*, and these are readily seen each to start with a distinct terminal.

For Rule 2 we must look for the "nullable" sections. There are three of these - in the productions for *EBNF0, Expression* and *Term*.

For *EBNF0*
The FIRST set of the nullable section is simply { nonterminal } and the FOLLOW set is { EOF }.

For *Expression*
The FIRST set of the nullable section is { "|" } and the FOLLOW set is FOLLOW(*Expression*), which is easily seen to be { "." , ")" , "]" , "}"

For *Term*
The FIRST set of the nullable section is FIRST(*Factor*) = { nonterminal, terminal "(" , "[" , "{" } and the FOLLOW set is FOLLOW(*Term*) = { "|" , ")" , "]" , "}" , "." } .

(b) The `nonterminal` definition allows a non-terminal to end with a lowline - as in `my_name__`. How should the definition be changed to allow lowlines within a non-terminal, but not at the end?

```
nonterminal = letter { { lowline } (letter | digit ) } .
```

c) Now consider a set of productions specifying simple English sentences:

```
Sentence      = "the" QualifiedNoun Verb | Pronoun Verb .
QualifiedNoun = [ Adjective { Adjective } ] Noun .
Noun          = "man" | "girl" | "boy" | "lecturer" .
Pronoun       = "he" | "she" .
Verb          = "talks" | "listens" | "mystifies" .
Adjective     = "tall" | "thin" | "sleepy" .
```

The productions leading on from *Sentenc*e allow a *QualifiedNoun* to consist of a *Noun* preceded by zero or more *Adjectives*. But in place of the line reading

```
QualifiedNoun = [ Adjective { Adjective } ] Noun .
```

one might have been tempted to write

```
QualifiedNoun = [ { Adjective } Adjective ] Noun .
```

or even

```
QualifiedNoun = { Adjective } Noun .
```

Do any of these suggestions introduce either ambiguity or non-LL(1) compliance into the grammar for *Sentence*? If they do, does this mean that the EBNF parser that could be constructed by Coco/R from the Cocol code given earlier would be unable to recognise the productions defining a *Sentenc*e

QualifiedNoun, in all three suggestions, has nullable sections. In the first suggestion there are two nullable sections. For the longer one […] the FIRST set is FIRST(*Adjective*) and the FOLLOW set is FIRST(*Noun*), so no problems arise. For the shorter one { *Adjective* } the FIRST and FOLLOW sets are also FIRST(*Adjective*) and FIRST(*Noun*), and so no problems arise. Similarly, for the simpler third suggestion the FIRST and FOLLOW sets of the nullable section { *Adjective* } are FIRST(*Adjective*) and FIRST(*Noun*).

But the second suggesstion also has two nullable sections. For the longer one […] the FIRST set is FIRST(*Adjective*) and the FOLLOW set is FIRST(*Noun*) - once again no problems arise. But for the shorter one { *Adjective* } the FIRST set is FIRST(*Adjective*) and the FOLLOW set is also FIRST(*Adjective*). So this suggestion is non-LL-1, and the grammar would not be able to parse sentences with adjectives in them. (Convince yourself of this if it's not immediately obvious.)

Task 2 - Ambiguity

The following Cocol productions try to describe a sequence of Roman numbers between 1 and 10:

```
COMPILER Roman
/* Parse a list of Roman numbers in the range 1 ... 10 - grammar only
   P.D. Terry, Rhodes University, 2017 */

IGNORE  CHR(9) .. CHR(13)

PRODUCTIONS
   Roman      = OneNumber { "," OneNumber } "." .
   OneNumber  = StartI | StartV | StartX .
   StartI     = "I" [ [ "I" ] [ "I" ] | "V" | "X" ] .
   StartV     = "V" [ "I" ] [ "I" ] [ "I" ] .
   StartX     = "X" .
END Roman.
```

The grammar is both non-LL(1) *and* ambiguous - note that a non-LL(1) grammar does not have to be ambiguous, although an ambiguous grammar cannot be LL(1).

To show ambiguity it is necessary only to find (at least) one sentence that can be derived in two ways. For this grammar there are several examples - only one is needed. For the record, you could derive VI in three ways, VII in three ways, and II in two ways.

Rule 2 - the FIRST/FOLLOW condition for nullable sections is broken in the productions for *StartI* and *StartV* as should be very obvious.

AN LL(1) compliant, unambiguous grammar is easily constructed:

```
PRODUCTIONS
   Roman      = OneNumber { "," OneNumber } "." .
   OneNumber  = StartI | StartV | StartX .
   StartI     = "I" [ "I" [ "I" ] ] | "V" | "X" ] .
   StartV     = "V" [ "I" [ "I" [ "I" ] ] ] .
   StartX     = "X" .
END Roman.
```

Task 3 - 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 76). 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.
```

Several submissions suggested (but did not justify) that maybe this problem could be solved by using a context-sensitive set of productions (which would not be LL(1)). That may be possible - I must think about it some more. Context-sensitive grammars are awkward to work with!

Task 4 - 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 realize 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, the simple grammar

```
Goal = "first" "next" { "next" | "other" } "next" "last" "." .
```

is non-LL(1), but it is not ambiguous - you could only parse the string

```
first  next  next  next  last  .
```

in one way. This is a particularly simple grammar and it is hopefully easy to see that *any* valid string defined by it could only be parsed in one way.

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 6 Steam Radio

The original grammar had productions

```
Radio        = { TalkShow | NewsBulletin | "song" | "advert" } EOF .
NewsBulletin = "advert" NewsItem { NewsItem } [ Weather ] Filler .
NewsItem     =    "ANC" [ "SACP" ] | "protest" | [ "SACP" ] "ANC" | "zuma"
               | "Helen" "tweet" | Scandal | Accident .
TalkShow     = "host" { listener "host" } [ Filler ] .
Accident     = "collision" "claims" "another" number "lives" .
Scandal      = { "GuptaLeak" } "corruption"  | "Hlaudi" .
Filler       = "song" | "advert"  .
Weather      = { "snow" | "rain" | "cloudy" | "windy" } .
listener     = identifier .
```

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

If we rewrite the grammar without meta-brackets, and using right recursion where necessary we get something like:

```
Radio        = Programmes EOF .
Programmes   = Programme Programmes | ε .
Programme    = TalkShow | NewsBulletin | "song" | "advert" .
NewsBulletin = "advert" NewsItems OptWeather Filler .
NewsItems    = NewsItem NewsItems | ε .
OptWeather   = Weather | ε .
NewsItem     =   "ANC" OptSACP | "protest" | OptSACP "ANC" | "zuma"
               | "Helen" "tweet" | Scandal" | Accident .
OptSACP      = "SACP" | ε .
TalkShow     = "host" Exchanges OptFiller .
Exchanges    = Exchange Exchanges | ε .
OptFiller    = Filler | ε .
Exchange     = listener "host" .
Filler       = "song" | "advert" .
Scandal      = Leaks "corruption" | "Hlaudi" .
Leaks        = "GuptaLeak" Leaks | ε .
Accident     = "collision" "claims" "another" number "lives" .
Weather      = OneWeather Weather | ε .
OneWeather   = "snow" | "rain" | "cloudy" | "windy" .
```

Rule 1 is broken in only two places:

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

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

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

Programmes, NewsItems, OptWeather, Weather, OptSACP, Exchanges, Leaks, OptFiller

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

```
FIRST(OptSACP)    = { SACP }
FOLLOW(OptSACP)   = { music, advert, ANC, protest, zuma, corruption, malema, SACP, hlaudi, helen,
                      guptaleak, collision, 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 | "song" | "advert" } EOF .
NewsItem     =    "ANC" [ "SACP" ] | "protest" | [ "SACP" ] "ANC" | "zuma"
                | "Helen" "tweet" | Scandal | Accident .
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 ["SACP"] is nullable, and one quickly discovers that Rule 2 is broken. Similarly:

```
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 specious (that is, indefensible) reasons. Just because a grammar is not LL(1) is no guarantee that you will not be able to 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 needed it could have been derived as a possible 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. And we remove the [ ] brackets on *Weather*.

```
Radio      = { TalkShow | "song" | "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 problematic "SACP" 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" "SACP" "ANC" in two ways (either as ("ANC" "SACP") (ε "ANC") or as ("ANC" ε) ("SACP" "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 "SACP" in the same breath, but if we ever mention "SACP" 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" [ "SACP" ] | "protest" | "SACP" "ANC"
            | "zuma" | "Helen" "tweet" | Scandal | Accident .
```

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 "SACP" immediately following "ANC" would be bound to that story on "ANC", and not to one still to follow. So the sequence

"ANC" "SACP" "ANC" "zuma"

would be handled as syntactically equivalent to

("ANC" "SACP") ("ANC") ("zuma")

and not as

("ANC") ("SACP" "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". In the case of the "dangling else", the recursive descent parser works very well, and if one wants the opposite sort of effect, one can resort to coding the subsidiary statements within { statement braces }.

Some suggested that the production just be written even more simply

```
NewsItem  =   "ANC" | "protest" | "SACP" | "zuma" |  "Helem" "tweet" | Scandal | Accident .
```

but no - one could then have news bulletins where "SACP" is mentioned without mention of the "ANC" at all.

Another intriguing suggestion from one group amounted to writing productions

```
NewsItem  =  "ANC" [ SACPItem ] | "protest" | "zuma" |  "Helem" "tweet" | Scandal | Accident .
SACPItem  =  "SACP" [ "SACP" ] "ANC" .
```

and you might like to explore whether they had, at last found a perfect solution. Yet aonother submission solved the problem by introducing some new terminals:

```
NewsItem  =   "ANCSACP" | "SACPANC" | "ANC" | "protest" | "zuma" | Scandal | Accident .
```

which, I guess, solves the problem, though not as I had intended!

Incorporating the town names into the weather forecasts is very simple! Just take care to parenthesize the options for the weather - omitting these parentheses would have the effect of binding the town to "snow" only.

```
Weather      = { town ( "snow" | "rain" | "cloudy" | "windy" ) } .
town         = identifier .
```

Task 7 - 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).

The basic grammar (for the original form of PVM code) demonstrates some techniques that are easily missed. We must treat the ends of lines to be significant markers this time, and we must make provision for an escape sequence in the definition of a string. The various kinds of opcode must be distinguished, and it must be possible to have completely blank lines. Note that we have introduced `lf` as a singleton character set, and then defined the EOL token to be comprised of this character (we cannot use elements of the CHARACTERS section directly in the PRODUCTIONS section - you may have missed this point as I don't think I stressed it at the time).

```
COMPILER Assem1 $CN
/* Grammar for subset of PVM assembler language
   No labels - all addressing is absolute
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  control        = CHR(0) .. CHR(31) .
  printable      = ANY - control .
  inString       = printable - '"' .
  digit          = "0123456789" .
  lf             = CHR(10) .

TOKENS
  number         = digit { digit } .
  stringLit      = '"' { inString | '\"' } '"' .
  EOL            = lf .
```

```
COMMENTS FROM ";" TO lf

IGNORE control - lf

PRODUCTIONS
  Assem1         = { PVMStatement } EOF .
  PVMStatement   = [ number ] [ PVMInstruction ] SYNC EOL .
  PVMInstruction = TwoWord | OneWord | PrintString .
  TwoWord        = ( "LDA" | "LDC" [ "-" ] | "LDL" | "STL" | "DSP" | "BRN" | "BZE"  ) 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" ) .
  PrintString    = "PRNS" stringLit .
END Assem1.
```

To allow for alphanumeric labels requires a few small changes. It is best to split the opcodes into 4 groups - those that take no argument, those that can take only a numerical argument, those that can take a numerical argument or a label, and PRNS. Note that we can have "standalone" labels too - in other words, lines with nothing but a label are permissible.

We have allowed the statements to incorporate either alphanumeric labels or the (redundant) numbers or both. As a simple exercise, consider how might allow one or the other, but *not* both.

```
TOKENS
  identifier     = letter { letter | digit } .
  number         = digit { digit } .
  stringLit      = '"' { inString | '\"' } '"' .
  EOL            = lf .

PRODUCTIONS
  Assem2         = { PVMStatement } EOF .
  PVMStatement   = [ number ] [ Label ] [ PVMInstruction ] SYNC EOL .
  PVMInstruction = TwoWord | OneWord | PrintString | Branch .
  TwoWord        = ( "LDA" | "LDC" [ "-" ] | "LDL" | "STL" | "DSP" ) 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" ) .
  Branch         = ( "BRN" | "BZE" ) ( number | Label ) .
  PrintString    = "PRNS" stringLit .
  Label          = identifier .
END Assem2.
```

A description of the `.COD` format is complicated only in that we have to allow for superfluous blank lines at the start and end of the program.

```
PRODUCTIONS
  Assem3         = { EOL } "ASSEM" EOL { EOL} "BEGIN" EOL { CODStatement } "END" "." { EOL } .
  CODStatement   = [ "{" number "}" CODInstruction ] SYNC EOL .
  CODInstruction = TwoWord | OneWord | PrintString .
  TwoWord        = ( "LDA" | "LDC" [ "-" ] | "LDL" | "STL" | "DSP" | "BRN" | "BZE"  ) 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" ) .
  PrintString    = "PRNS" stringLit .
END Assem3.
```

Allowing for a system that will accept either the original `.pvm` format or the `.cod` format (but not intermingled) is a little tricky if one is not to insist that the distinguishing `ASSEM` directive come immediately at the start of the file. The solution suggested below actually suffers from an LL(1) violation, but (like the dangling else situation) in this case it is of no consequence - the parser would simply mop up leading EOL tokens until a point is reached where the distinction can be made.

```
PRODUCTIONS
  Assem4         = { EOL } ( PVMFormat | CODFormat ) .

  PVMFormat      = { PVMStatement } EOF .
  PVMStatement   = [ number ] [ Label ] [ PVMInstruction ] SYNC EOL .
  PVMInstruction = OneWord | TwoWord | PrintString | PVMBranch .
  PVMBranch      = ( "BRN" | "BZE" ) ( number | Label ) .
```

```
  CODFormat      = "ASSEM" EOL { EOL} "BEGIN" EOL { CODStatement } "END" "." { EOL }.
  CODStatement   = [ "{" number "}" CODInstruction ] SYNC EOL .
  CODInstruction  = OneWord | TwoWord | PrintString | CODBranch .
  CODBranch      = ( "BRN" | "BZE" ) number .

  TwoWord        = ( "LDA" | "LDC" [ "-" ] | "LDL" | "STL" | "DSP" ) 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" ) .
  PrintString    = "PRNS" stringLit .
  Label          = identifier .
END Assem4.
```

One might have been tempted to introduce a `comment` token, and then modified the grammar on the following lines

```
CHARACTERS
  inComment = ANY - control .
TOKENS
  comment    = ";" { inComment } .
PRODUCTIONS
  PVMStatement   = [ number ] [ Label ] [ PVMInstruction ] [ comment ] SYNC EOL .
  CODStatement   = [ "{" number "}" CODInstruction ] [ comment ] SYNC EOL .
```

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

This is, incidentally, a system where the `IGNORECASE` directive in Coco/R might usefully be applied.

Finally note that, while it may be common to find `HALT` as the last statement in a program, this is not demanded. As an example:

```
                            ; read a sentence up to a period
        DSP   1             ; char ch;
WHILE   LDA   0             ; while (true) {
        INPC                ;   Read(ch)
        LDL   1
        LDC   46            ;   if (ch == '.') halt;
        CEQ
        BZE   PRINT
        HALT
PRINT   LDL   0
        PRNC                ;   write(ch)
        BRN   WHILE         ; } // while(true)
```

Task 5 - All very logical

The Boolean grammar is very similar to the arithmetic one, of course, so it was distressing that so many people had it rather badly wrong. To get the operator precedences correct, while keeping the grammar LL(1), it is easiest to introduce four levels of non-terminal:

```
COMPILER Bool $CN
/* Boolean expression parser
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
TOKENS
  variable   = letter .

COMMENTS FROM "(*" TO "*)"  NESTED
COMMENTS FROM "/*" TO "*/"  NESTED
IGNORE CHR(0) .. CHR(31)
```

```
PRODUCTIONS
  Bool       = { Expression "=" } .
  Expression = Term { Or Term } .
  Term       = Factor { [ And ] Factor } .
  Factor     = "NOT" Factor | Primary { "'" } .
  Primary    = True | False | variable | "(" Expression ")" .
  True       = "TRUE" | "1" .
  False      = "FALSE" | "0" .
  And        = "AND" | "&" | "." .
  Or         = "OR" | "|" | "+" .
END Bool.
```

Many people wrote the constants and operators into the TOKENS section as follows:

```
TOKENS
  variable   = letter .
  True       = "TRUE" | "1" .
  False      = "FALSE" | "0" .
  And        = "AND" | "&" | "." .
  Or         = "OR" | "|" | "+" .
```

but it is usually adequate to introduce fixed token definitions straight into the PRODUCTIONS section where they appear, and reserve the TOKENS section for the definition of things like identifiers, numbers and strings that are all similar in form but different in spelling.