Computer Science 3 - 2015

Programming Language Translation

Practical for Week 3, beginning 14 September 2015 - Solutions

Complete sources to these solutions can be found on the course WWW pages in the files `PRAC3A.ZIP`.

Task 2 - Extensions to the Simple Calculator

In the source kit you were given `Calc.atg`. This is essentially the calculator grammar on page 62 of the textbook, and you were invited to extend it to allow for parentheses, leading unary + or -operators, an abs() funcion, a factorial capability, numbers with decimal points and so on.

Extending the calculator grammar can be done in several ways. Here is one of them, which corresponds to the approach taken to expressions in languages like Pascal, which do not allow two signs to appear together:

```
COMPILER Calc1  $CN
/* Simple four function calculator (extended)
   P.D. Terry, Rhodes University, 2015 */

CHARACTERS
  digit      = "0123456789" .
  hexdigit   = digit + "ABCDEF" .

TOKENS
  decNumber  =   digit { digit } [ "." { digit } ]
               | "." digit { digit } .
  hexNumber  =   "$" hexdigit { hexdigit } .

IGNORE CHR(0) .. CHR(31)

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

Another approach, similar to that taken in C++, is as follows:

```
PRODUCTIONS
  Calc2      = { Expression "=" } EOF .
  Expression = Term { "+" Term  |  "-" Term } .
  Term       = Factor { "*" Factor |  "/" Factor } .
  Factor     = ( "+" | "-") Factor | Primary { "!" } [ "↑" Factor ] .
  Primary    = decNumber | hexNumber | "(" Expression ")" | "abs" "(" Expression ")" .
END Calc2.
```

This allows for expressions like 3 + - 7 or even 3 * -4 or even 3 / + - 4. Because of the way the grammar is written, the last of these is equivalent to 3 / (+ (- (4))).

Here are some other attempts. What, if any, differences are there between these and the other solutions presented so far?

```
PRODUCTIONS
  Calc3      = { Expression "=" } EOF .
  Expression = ["+" | "-" ] Term { "+" Term  | "-" Term } .
  Term       = Factor { "*" Factor | "/" Factor } .
  Factor     =   Primary { "!" }
               |  "abs" "(" Expression ")" .
  Primary    = decNumber | hexNumber | "(" Expression ")" .
END Calc3.


PRODUCTIONS
  Calc4      = { Expression "=" } EOF .
  Expression = Term { "+" Term  | "-" Term } .
  Term       = Factor { "*" Factor | "/" Factor } .
  Factor     =   ( "+" | "-" ) ( Factor | Primary )
               | "abs" "(" Expression ")" ) .
  Primary    = decNumber | hexNumber | "(" Expression ")" { "!" } .
END Calc4.
```

It may be tempting to suggest a production like this

```
Primary    =  decNumber | hexNumber | "(" Expression ")" | "abs(" Expression ")" .
```

However, a terminal like `"abs("` is restrictive. It is invariably better to allow white space to appear between method names and parameter lists if the user prefers this style.

Several submissions tried to define a number token to incorporate an optional sign. While this is used as an illustration in The Book, it is not the best way of doing it when one is trying to describe free-format expressions, where one might like to separate leading + and - signs from the numbers that might follow them. Furthermore, describing numbers in the PPRODUCTIONS section, for example as

```
Number = decNumber [ "." decNumber ] .
```

is not a good idea either. Numbers with points in them are nearly always written as contiguous characters, hence 3.45 and not 3 . 45. Remember that in Cocol spaces may be inserted between tokens (but very rarely within tokens, save when these are bracketed by unique delimiters such as quotes). And an even worse idea is

```
Number =  { digit } "." { digit } .
```

which would allow numbers to contain nothing more than a decimal point!

Task 3 - Meet the family

This was meant to be relatively straightforward and should not have caused too many difficulties. A criticism of several submissions was that they were too restrictive. Here is one solution in the spirit of the exercise:

```
COMPILER Family1 $CN
/* Describe a family
   P.D. Terry, Rhodes University, 2015 */

CHARACTERS
  control       = CHR(0) .. CHR(31) .
  uLetter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter       = "abcdefghijklmnopqrstuvwxyz" .
  digit         = "0123456789" .

TOKENS
  name          = uLetter { lLetter | "'" uLetter | "-" uLetter } .
  number        = digit { digit } .

IGNORE control

PRODUCTIONS
  Family1       = { Section SYNC } EOF .
  Section       = Surname | Parents | Grandparents | Children | Grandchildren | Possession .
  Surname       = "Surname" ":" name { name } .
  Parents       = "Parents" ":" PersonList .
  Grandparents  = "Grandparents" ":" PersonList .
  Children      = "Children" ":" PersonList .
  Grandchildren = "Grandchildren" ":" PersonList .
  PersonList    = OnePerson { "," OnePerson } .
  OnePerson     = name { name } [ "(" "deceased" ")" ] { Description } [ Spouse ] .
  Spouse        = "=" name { name } .
  Description   = "[" Relative "of" OnePerson "]" .
  Relative      =   "son"  | "daughter" | "mother"  | "father"
                  | "wife" | "husband"  | "partner" | "mistress" .
  Possession    = number [ "small" | "large" ]
                          (   "cat"   | "cats"   | "dog" | "dogs" | "bagpipe" | "bagpipes"
                            | "house" | "houses" | "car" | "cars" )  .
END Family1.
```

That solution does not insist that the surname should be part of all descriptions. Here is an alternative PRODUCTIONS set that does just that, and also factorizes the grammar slightly differently:

```
PRODUCTIONS
  Family2       = { Generation SYNC } Surname SYNC { Generation  SYNC } { Possession }  EOF .
  Surname       = "Surname" ":" name { name } .
  Generation    = ( "Parents" | "Grandparents" | "Children" | "Grandchildren" ) ":" PersonList .
  PersonList    = OnePerson { "," OnePerson } .
  OnePerson     = name { name } [ "(" "deceased" ")" ] { Description } [ Spouse ] .
```

```
  Spouse        = "=" name { name } .
  Description   = "[" Relative "of" OnePerson "]" .
  Relative      =   "son"  | "daughter" | "mother"  | "father"
                  | "wife" | "husband"  | "partner" | "mistress" .
  Possession    = number [ "small" | "large" ]
                         (   "cat"   | "cats"   | "dog" | "dogs" | "bagpipe" | "bagpipes"
                           | "house" | "houses" | "car" | "cars" )  .
END Family2.
```

Four points are worth making (a) the `Surname` section should not have allowed the possibility of listing the name as deceased (b) it is better to use a construct like `"(" "deceased" ")"` than `"(deceased)"` as a single terminal (c) relationships are best between `OnePerson` and another `OnePerson`, and not simply between `OnePerson` and some names (d) there is no need to make line feeds significant in this example - although no harm is done if you do, and they certainly make the text easier for a human reader to decode.

Note how we have defined "cat" and "cats" as keywords. We might alternatively have introduced a token

```
item = lLetter { lLetter } .
```

and changed the production to allow for all sorts of other goodies!

```
Possession = number [ "small" | "large" ] item .
```

Task 4 - One for the musicians in our midst

The exercise called for you to develop a Cocol grammar that describes the words of a song and the notes sung to those words, expressed in "Tonic Solfa".

This toy problem is straightforward, but note the way in which an `lf` singleton character set is introduced from which the single character `EOL` token is defined - this is a rather unusual case (in most languages end-of-line is insignificant). Note also that a line of words might also contain some tonic solfa key words as ordinary words - for example "so" and "me". Note how the token `word` has been defined - multiple - and ' characters are allowed, but at most one trailing punctuation mark. We probably would not want to cater for sequences like `Tom!!,Dick,Harry` as making up one word.

It is preferable to use CHR(10) = `lf` as the line mark and to ignore CHR(13) = `cr`. Then the system will work equally well on Windows and on Linux systems. On a Mac, just to be perverse, they choose to use a single `cr` character to mark line breaks. You might like to decide how one could define a line feed token that could suffice on all three operating systems.

```
COMPILER Solfa $CN
/* Describe the words and notes of a tune using tonic solfa
   P.D. Terry, Rhodes University, 2015 */

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

TOKENS
  word       = letter { letter | "'" | "-" letter } [ "." | "," | "!" | "?" ] .
  EOL        = lf .

IGNORE control - lf

PRODUCTIONS
  Solfa      = { Line } EOF .
  Line       = Words EOL Tune EOL EOL { EOL } .
  Words      = ( word | Note ) { word | Note } .
  Tune       = Note { Note } .
  Note       = "do" | "re" | "me" | "fa" | "so" | "la" | "te" .
END Solfa.
```

There are, in fact, other note names in Tonic Solfa, which can be handled in the same way.

These "words" were pretty minimal. One might have wanted to have lines in a song like

"Ha ha" said the clown

(you are too young to remember that one, I bet - about 1967). To handle this one might describe a word as having an alternative that could be in the form of a string (similar to strings in Parva). Fill in the details for yourselves.

Task 5 - Golden Oldies - hits of a bygone era

The trick here, once again, is to define appropriate character sets and tokens, when it all becomes very easy:

```
COMPILER Nostalgia $CN
/* Describe a list of pop songs along with performers and dates of release
   They don't write them like they used to do!
   P.D. Terry, Rhodes University, 2015 */

CHARACTERS
  control  = CHR(0) .. CHR(31) .
  inTitle  = ANY - control - '"' .
  inArtist = ANY - control - "()" .
  digit    = "0123456789" .

TOKENS
  title  = '"' inTitle { inTitle } '"' .
  artist = "(" inArtist { inArtist } ")" .
  year   = digit digit digit digit .

IGNORE control



PRODUCTIONS
  Nostalgia = { PopTune } EOF.
  PopTune   = title artist { "," artist } [ "[" year "]" ] SYNC "." .
END Nostalgia.
```

There are, of course, other ways to try to do this. Since the tokens are clearly demarcated by quotes at each end or by parentheses at each end, it makes some sense to treat `title` and `artist` as simple tokens - which, unusually for most computer languages, can contain embedded spaces - as well as all sorts of other printable characters, perhaps, like single quotes, digits, hyphens and the like.

Since a year will have exactly four digits, it is best specified as above. The examples showed a variety of ways of adding the year, with suggestive spaces - for example [1963] and [1962] - so that it is preferable to leave the `year` token as the digit part, and add the enclosing [ ] brackets as shown above, rather than defining, for example

```
TOKENS
  year   = "[" digit digit digit digit "]" .
```

One might be tempted to write a grammar with productions like

```
PRODUCTIONS
  Nostalgia = { PopTune } EOF.
  PopTune   = title artist [ "[" year "]" , { "[" year "]" } ] SYNC "." .
END Nostalgia.
```

but the examples suggested that a tune could be recorded by several artists on one year, rather than by one artist in several years. Come to think of it, a better way of recording tunes would be to record the artist along with the year, leading to a grammar like

```
PRODUCTIONS
  Nostalgia = { PopTune } EOF.
  PopTune   = title Recording { "," Recording } SYNC "." .
  Recording = artist [ "[" year "]" ] .
END Nostalgia.
```

Would this grammar also describe the sample list of tunes given in the source kit? If not, where not?

Task 6 - So what if Parva is so restrictive - fix it!

The Parva extensions produced some interesting submissions. Many of them (understandably!) were too restrictive in certain respects, while others were too permissive. Admittedly there is a thin line between what might be "nice to have" and what might be "sensible to have" or "easy to compile". Here is a heavily commented suggested solution, incorporating the extra exercises added for Prac 4.

```
COMPILER Parva $CN
/* Parva level 2.1 grammar  - Coco/R for C# - Practical 3
   P.D. Terry, Rhodes University, 2015
   Extended Grammar only */

CHARACTERS
  lf           = CHR(10) .
  backslash    = CHR(92) .
  control      = CHR(0) .. CHR(31) .
  letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .
  nonZeroDigit = "123456789" .
  binDigit     = "01" .
  hexDigit     = digit + "abcdefABCDEF" .
  stringCh     = ANY - '"' - control - backslash .
  charCh       = ANY - "'" - control - backslash .
  printable    = ANY - control .

TOKENS

/* Insisting that identifiers cannot end with an underscore is quite easy */

  identifier = letter { letter | digit | "_" { "_" } ( letter | digit ) } .

/* but a simpler version is what many people think of

  identifier = letter { letter | digit | "_" ( letter | digit ) } .

   Technically this is not quite what was asked.  The restriction is really that an
   identifier cannot end with an underscore.  Identifiers like Pat_____Terry are allowed:

*/

/* Allowing (and restricting) numbers to be of the various forms suggested is easy enough */

  number     = "0" | nonZeroDigit { digit } | digit { hexDigit } 'H' | binDigit { binDigit } '%' .

/* But be careful.  There is a temptation to define
      digit  = "123456789" .
      number = "0" | digit { digit | "0" } .
   and then forget that
      identifier = letter { letter | digit | "_" } .
   would not allow identifiers to have 0 in them */


  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" { charCh   | backslash printable } "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE control

PRODUCTIONS
  Parva                = { ConstDeclarations | FuncOrVarDeclarations } .

/* The options in Statement are easily extended to handle the new forms */

  Statement            = (  Block
                          | ConstDeclarations | VarDeclarations
                          | AssignmentOrCall
                          | IfStatement        | WhileStatement
                          | ReturnStatement    | HaltStatement
                          | ReadStatement      | WriteStatement
                          | ReadLineStatement  | WriteLineStatement
                          | ForStatement       | BreakStatement
                          | ContinueStatement  | RepeatStatement
                          | SwitchStatement    | ";"
                        ) .

  Block                = "{" { Statement } "}" .
```

```
/* Declarations remain the same as before */

  FuncOrVarDeclarations = Type identifier ( Function | GlobalVars ) .
  Function              = "(" FormalParameters ")" Block .

  FormalParameters      = [ OneParam { "," OneParam } ] .
  OneParam              = Type identifier .

  GlobalVars            = OneGlobal { "," identifier OneGlobal } ";" .
  OneGlobal             = [ "=" Expression ] .

  ConstDeclarations     = "const" OneConst { "," OneConst } ";" .
  OneConst              = identifier "=" Constant .
  Constant              = number | charLit | "true" | "false" | "null" .

  VarDeclarations       = Type OneVar { "," OneVar } ";" .
  OneVar                = identifier [ "=" Expression ] .

/* AssignmentOrCall statements require care to avoid LL(1) problems.
   To handle the "useful assignment operators" is easy! */

  AssignmentOrCall      = (   Designator (   AssignOp Expression
                                           | "++"
                                           | "--"
                                           | "(" Arguments ")"
                                         )


                            | "++" Designator
                            | "--" Designator
                          ) ";" .

   AssignOp             = "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "&=" | "|=" .

/* In all these it is useful to maintain generality by using Designator, not identifier */

  Designator            = identifier [ "[" Expression "]" ] .

  Arguments             = [ OneArg { "," OneArg } ] .
  OneArg                = Expression .

/* The if-then-elsif-else construction is most easily described as follows. Although
   this is not LL(1), this works admirably - it is simply the well-known dangling
   else ambiguity, which the parser resolves by associating elsif and else clauses
   with the most recent if */

 IfStatement         = "if" "(" Condition ")" Statement
                       { "elsif" "(" Condition ")" Statement }
                       [ "else" Statement ] .


/* The switch statement has to be handled carefully.  The labelled case "arms" are
   optional, but the "default" option can only appear once.  The selection can best
   be done by a general Expression rather than an identifier, and the case "labels"
   can be constants of any type that would match the type of the selector expression,
   including identifiers defined in a ConstDeclarations section */

  SwitchStatement       = "switch" "(" Expression ")" "{"
                            { OneCase }
                            [ "default" ":" { Statement } ]
                          "}" .
  OneCase               = CaseLabel ":" { Statement } .
  CaseLabel             = "case" ( Constant | ("+" | "-") ( number | ConstantIdentifier ) ) .
  ConstantIdentifier    = identifier .

/* You might like to consider the differences (if any) between the preceding definition of a
   switch statement and the alternative below

  SwitchStatement       = "switch" "(" Expression ")" "{"
                            { CaseLabelList Statement { Statement } }
                            [ "default" ":" { Statement } ]
                          "}" .
  CaseLabelList         = CaseLabel { CaseLabel } .
  CaseLabel             = "case" [ "+" | "-" ] ( Constant | Constantidentifier ) ":" .
*/

/* The case arms/clauses usually have to contain a "break" statement, which is syntactically
   simply another form of statement.  There is actually a context-sensitive feature
   embedded in this - break statements cannot really be placed "anywhere", but we
   reserve further discussion for a later occasion. */
```

```
/* Remember that the RepeatStatement must end with a semicolon - easy to forget this!
   Unlike the C family's DoWhile, we can have a sequence of statements in the loop body */

  RepeatStatement       = "repeat" { Statement } "until" "(" Condition ")" ";" .

/* The Modula-2 inspired ForStatement needs to avoid using AssignmentStatement as one might be tempted to do.
   It is sensible to control a for loop using a simple identifier rather than a general
   Designator  for reasons that might be discussed later in the course.
   In fact, the "by" form was used with "to" in Modula-2, while the "to"/"downto" form
   (without "by") was used in Pascal.  So what we have here is even more general!
   Note that by clever factoring we can accept both the Modula-2 and Python variations */

  ForStatement          = "for" identifier
                          (  "in" ExpList
                           | "=" Expression ( "to" | "downto" ) Expression [ "by" Expression ]
                          )
                          Statement .

/* Break and Continue statements are very simple.  They are really "context dependent" but we
   cannot impose such restrictions in a context free grammar */

  BreakStatement        = "break" ";" .
  ContinueStatement     = "continue" ";" .

/* ReadLine and WriteLine statements must allow for an empty argument list */

  ReadLineStatement     = "readLine" "(" [ ReadElement { "," ReadElement } ] ")" ";" .
  WriteLineStatement    = "writeLine" "(" [ WriteElement { "," WriteElement } ] ")" ";" .

/* Much of the rest of the grammar remains unchanged: */

  WhileStatement        = "while" "(" Condition ")" Statement .
  ReturnStatement       = "return" [ Expression ] ";" .
  HaltStatement         = "halt" ";" .
  ReadStatement         = "read" "(" ReadElement { "," ReadElement } ")" ";" .
  ReadElement           = stringLit | Designator .
  WriteStatement        = "write" "(" WriteElement { "," WriteElement } ")" ";" .
  WriteElement          = stringLit | Expression .
  Condition             = Expression .

/* The basic form of Expression introduces "in", effectively as another relational operator
   with the same precedence as the other relational operators */

  Expression            = AddExp [ RelOp AddExp | "in" ExpList ] .
  AddExp                = [ "+" | "-" ] Term { AddOp Term } .
  Term                  = Factor { MulOp Factor } .
  Factor                =   Designator [ "(" Arguments ")" ]
                          | Constant
                          | "new" BasicType "[" Expression "]"
                          | "!" Factor

/* Type conversion functions are easy to add syntactically, but as an illustration
   we have not used the (type) casting syntax as found in the C family.
   A function should be notated as a function! */

                          | [ "char" | "int" ] "(" Expression ")" .

/* The infix operators here are grouped in three levels */

  AddOp                 = "+" | "-" | "||" .
  MulOp                 = "*" | "/" | "%" | "&&" .
  RelOp                 = "==" | "!=" | "<" | "<=" | ">" | ">=" .

/* *************************

  To handle Expressions with C#-inspired precedence we might proceed as follows.
  Note the use of the { } metabrackets in all but one of the following. (Why?)
  Type conversion functions in this mode are a bit tricky to add syntactically,
  that is, if we are to use the (type) casting syntax as found in the C family.
  We also have to split the various operators into different groups.

  Expression            = AndExp { "||" AndExp } .
  AndExp                = EqlExp { "&&" EqlExp } .
  EqlExp                = RelExp { EqlOp RelExp } .
  RelExp                = AddExp [ RelOp AddExp | "in" ExpList ] .  // [ ] not { }
  AddExp                = MulExp { AddOp MulExp } .
  MulExp                = Factor { MulOp Factor } .
  Factor                = Primary | ( "+" | "-" | "!" ) Factor .
  Primary               =   Designator [ "(" Arguments ")" ]
                          | Constant
                          | "new" BasicType "[" Expression "]"
```

```
                          | "(" (   "char" ")" Factor
                                | "int"  ")" Factor
                                |  Expression ")"
                              ) .

  AddOp                = "+" | "-" .
  MulOp                = "*" | "/" | "%" .
  EqlOp                = "==" | "!=" .
  RelOp                = "<" | "<=" | ">" | ">=" .

**************************** */

 /* The ExpList used after the "in" operator can be quite general, syntactically */

  ExpList              = "{" Range { "," Range } "}" .
  Range                = Expression [ ".." Expression ] .

  Type                 = "void" | BasicType [ "[" "]" ] .

 /* char is simply added as an optional BasicType */

  BasicType            = "int" | "bool" | "char" .

 END Parva.
```

Task 8 - 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.

```
COMPILER Assem1 $CN
/* Grammar for subset of PVM assembler language
   No labels - all addressing is absolute
   P.D. Terry, Rhodes University, 2015 */

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.
```

When I set this some years ago I challenged the class to extend the system so that it could accept programs written in either the PVM or the COD format (which the assembler used in prac 2 could do, as you might have realised). Although this was not asked of you this year, it may be of interest to show the extended solution as well.

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.

Finally, note that may be usual to find `HALT` as the last statement in a program, this is not demanded.