Computer Science 3 - 2016

Programming Language Translation

Practical for Week 3, beginning 14 September 2016 - 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 and numbers with decimal points.

```
COMPILER Calc1  $CN
/* Simple four function calculator (extended)
   P.D. Terry, Rhodes University, 2016 */

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     = decNumber | hexNumber | "(" Expression ")" .
END Calc1.
```

Task 3 - Take a logical approach to your practicals

A simple grammar for logical expressions is similar in some respects to the one in Task 2

```
COMPILER Bool1  $CN
/* Simple Boolean expression evaluator
   P.D. Terry, Rhodes University, 2016 */

CHARACTERS
  digit      = "0123456789" .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
               + "abcdefghijklmnopqrstuvwxyz" .

TOKENS
  identifier = letter { letter | digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Bool       = { identifier "=" Expression ";" } EOF .
  Expression = Term   { ( "or"  | "||" ) Term   } .
  Term       = Factor { ( "and" | "&&" ) Factor } .
  Factor     =   ( "not" | "!" ) Factor
               | identifier | "true" | "false" | "(" Expression ")" .
END Bool1.
```

Task 4 - 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.

```
COMPILER Parva $CN
/* Parva level 1 grammar  - Coco/R for C# (EBNF)
   P.D. Terry, Rhodes University, 2016
   Extended Grammar for Prac 3 - solution */

CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  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: */

  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"
IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva             = "void" identifier "(" ")" Block .
  Block             = "{" { Statement } "}" .

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

  Statement         =    Block
                       | ConstDeclarations | VarDeclarations
                       | Assignments
                       | IfStatement       | WhileStatement
                       | ReturnStatement   | HaltStatement
                       | ReadStatement     | WriteStatement
                       | ForStatement      | BreakStatement
                       | ContinueStatement | DoWhileStatement
                       | ";" .

/* Declarations remain the same as before */

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

/* We can introduce the extra form of asssignment operators as tokens as follows.  Note
   theta we do not want to use, say "+" "=" because they cannot contain spaces. */

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

/* Don't be tempted to use the CompoundAssignOp in the declaration of OneVar or OneConst.
   It cannot have a proper semantic meaning if you do.  All you can use is "=" */

  OneVar            = identifier [ "=" Expression ] .

/* One way of introducing the extended form of assignment statements might be to define

  Assignments = Designator { "," Designator } ( "=" | CompoundAssignOp )
                Expression { "," Expression } .

*/

/* However, we might wish to limit the form with multiple Designators and Expressions to
   use only the simple = operator, for ease of code generation as we may see later.

   If this is the case, to keep thing LL(1) compliant, the extra forms of assignment
   statement familiar from Python, and the compound assignments familiar from C and C# are
   best handled as below.  THis affords a nice example of factoring a grammar to avoid
   LL(1) conflicts by delaying the awkward component for a while */

  Assignments       = Designator
                      (   CompoundAssignOp Expression
                        | { "," Designator } "=" Expression { "," Expression }
                      ) ";" .
```

```
    Designator        = identifier [ "[" 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 ] .

    WhileStatement    = "while" "(" Condition ")" Statement .

  /* Remember that the DoWhile statement must end with a semicolon - easy to forget this!
     Why don't we need a semicolon as a terminator for a WhileStatement or IfStatement? */

    DoWhileStatement  = "do"  Statement  "while" "(" Condition ")" ";" .

  /* Break and Continue statements are very simple.  They are really "context dependent" but we
     cannot impose such restrictions in a context-free grammar.  And they also need their own
     terminating semicolons, which tend to be forgotten. */

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

    ReturnStatement   = "return"   ";" .
    HaltStatement     = "halt"     ";" .

  /* The Pascal 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.  And note
     that the initial and final values of the controlling variable are most generally allowed
     to be Expressions rather than simple constants or variables.

     But don't be tempted to use OneVar either!  OneVar is a declaration kind of statement
     really,  As we shall see, it is compiled in a rather different way. */

    ForStatement      = "for" identifier "=" Expression ( "to" | "downto" ) Expression
                        Statement .

    ReadStatement     = "read" "(" ReadElement { "," ReadElement } ")" ";" .
    ReadElement       = stringLit | Designator .
    WriteStatement    = "write" "(" WriteElement { "," WriteElement } ")" ";" .
    WriteElement      = stringLit | Expression .
    Condition         = Expression .
    Expression        = AddExp [ RelOp AddExp ] .
    AddExp            = [ "+" | "-" ] Term { AddOp Term } .
    Term              = Factor { MulOp Factor } .
    Factor            =   Designator | Constant
                        | "new" BasicType "[" Expression "]"
                        | "!" Factor | "(" Expression ")" .
    Type              = BasicType [ "[]" ] .
    BasicType         = "int" | "bool" .
    AddOp             = "+" | "-" | "||" .

  /* The % operator is easily added to the set of MulOps */

    MulOp             = "*" | "/" | "&&" | "%" .

    RelOp             = "==" | "!=" | "<" | "<=" | ">" | ">=" .
  END Parva.
```

Task 5 - One for the Musicians in our Midst (but the rest of you should do it too)

The solutions received were mixed. Some didn't capture the idea that there should be N competitions each with M bands. A typical over simplified attempt looks like this

```
    Gath1         = { Competition } .
    Competition   = SlowQuick |  MSR |  Medley .
    SlowQuick     = "SlowMarch" "March"  .
    MSR           = "March" "Strathspey" "Reel" [ "March" ] .
    Medley        = OneTune { OneTune } .
    OneTune       =   "March" | "SlowMarch" | "Jig" | "Hornpipe" | "Reel"
                    | "Strathspey" { "Strathspey" } "Reel" .
```

If we want to introduce the idea that there are multiple competitions with multiple bands, then it might seem to make sense to try

```
  Gath2       = { Competition } .
  Competition = Band { Band } .
  Band        = SlowQuick | MSR | Medley .
  SlowQuick   = "SlowMarch" "March" .
  MSR         = "March" "Strathspey" "Reel" [ "March" ] .
  Medley      = OneTune { OneTune } .
  OneTune     =   "March" | "SlowMarch" | "Jig" | "Hornpipe" | "Reel"
                | "Strathspey" { "Strathspey" } "Reel" .
```

This is badly non LL(1) (If you think about it, it represents a continual wail of noise). And it doen't capture the idea that every band in a particular competitio must play the same kind of selectio if tunes. So, for a start, we need to get some sort of break between bands at least:

```
  Gath3       = { "AnnounceCompetition" Competition } .
  Competition = Band { Band } .
  Band        = "AnnounceBand" ( SlowQuick | MSR | Medley ).
  SlowQuick   = "SlowMarch" "March" "break" .
  MSR         = "March" "Strathspey" "Reel" [ "March" ] "break" .
  Medley      = OneTune { OneTune } "break" .
  OneTune     =   "March" | "SlowMarch" | "Jig" | "Hornpipe" | "Reel"
END Gath3.
```

Even this is non-LL(1). We can get an LL(1) grammar if we make the right kind of announcements (compare the production for *Band* with a prduction for Parva where most statement kinds start with a unique key word):

```
  Gath4       = { "AnnounceCompetition" Competition } .
  Competition = Band { Band } .
  Band        =   "AnnounceSlow"  SlowQuick
                | "AnnounceMSR" MSR
                | "AnnounceMedley" Medley .
  SlowQuick   = "SlowMarch" "March" "break" .
  MSR         = "March" "Strathspey" "Reel" [ "March" ] "break" .
  Medley      = OneTune { OneTune } "break" .
  OneTune     =   "March" | "SlowMarch" | "Jig" | "Hornpipe" | "Reel"
                | "Strathspey" { "Strathspey" } "Reel" .
```

But none of the above capture the idea that all the bands in one competition must play the same kind of "set" (as they are called). Here is a much better solution that, at last, does that too:

```
COMPILER Gath5 $CN
/* Describes the Pipe Band events at a Highland Gathering
   P.D. Terry, Rhodes University, 2016 */

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS

  Gath5         = { Competition } .
  Competition   =   "AnnounceSlow"   SlowQuickComp
                  | "AnnounceMSR"    MSRComp
                  | "AnnounceMedley" MedleyComp .
  SlowQuickComp = SlowQuick { SlowQuick } .
  SlowQuick     = "SlowMarch" "March" "break" .
  MSRComp       = MSR { MSR } .
  MSR           = "March" "Strathspey" "Reel" [ "March" ] "break" .
  MedleyComp    = Medley { Medley } .
  Medley        = OneTune { OneTune } "break" .
  OneTune       =   "March" | "SlowMarch" | "Jig" | "Hornpipe" | "Reel"
                  | "Strathspey" { "Strathspey" } "Reel" .

END Gath5.
```

In all these notice that we have not fallen into the trap of defining:

```
  OneTune     =   "March" | "SlowMarch" | "Jig" | "Hornpipe" | "Reel"
                | "Strathspey" { "Strathspey" } "Reel" { "Reel" } .
```

which would make the grammar non-LL(1) again (do you see why - check RUle 2), or of trying to write

```
  OneTune     =   "March" | "SlowMarch" | "Jig" | "Hornpipe" |
                | "Strathspey" { "Strathspey" } "Reel" { "Reel" } .
```

which is LL(1), but prevents reels being played without a strathspey immediately before them.

Task 6 - Meet some of the staff of the Hamilton Building

This can be attempted in several ways. As always, it is useful to try to introduce non-terminals for the items of semantic interest. Here is one attempt at a solution:

```
COMPILER Staff1 $CN
/* Describe a list of staff in a department
   Non-LL(1), and will not work properly
   P.D. Terry, Rhodes University, 2016 */

CHARACTERS
  uLetter  = "ABCDEFGHIJKLMNOPQRSTUVWZYZ" .
  lLetter  = "abcdefghijklmnopqrstuvwzyz" .
  letter   = uLetter + lLetter .

TOKENS
  name     = uLetter { letter | "'" uLetter | "-" uLetter } .
  initial  = uLetter "." .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Staff1       = { Person } EOF .
  Person       = [ Title ] { name | initial } surname { "," Degree } SYNC "." .
  Title        = "Professor" | "Prof" | "Dr" | "Mr" | "Mrs" | "Ms" | "Mx" | "Miss" .
  surname      = name .
  Degree       =   "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci"
                 | "BSc(Hons)" | "BCom(Hons)" | "MA" | "MSc" | "PhD" .
END Staff1.
```

Note that this allows for names like Smith and Malema and also for names like MacKay, O'Neill and Lee-Ann.

Although this correctly describes a staff list, it is useless for a simple parser, as surnames and names are lexically indistinguishable. This is another example of an LL(1) violation.

Attempting to define a better grammar affords interesting insights. It is not difficult to come up with productions that permit full names with leading initials (only) or to contain complete names only:

```
PRODUCTIONS
  Staff2       = { Person } EOF .
  Person       = [ Title ] FullName { "," Degree } SYNC "."  .
  FullName     = InitialsName | name { name } .
  InitialsName = initial { initial } name .
  Title        = "Professor" | "Prof" | "Dr" | "Mr" | "Mrs" | "Ms" | "Mx" | "Miss" .
  Degree       =   "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci"
                 | "BSc(Hons)" | "BCom(Hons)" | "MA" | "MSc" | "PhD" .
END Staff2.
```

but this is too restrictive. Another attempt might drop the distinction between initials and complete names:

```
PRODUCTIONS
  Staff3       = { Person } EOF .
  Person       = [ Title ] FullName { "," Degree } SYNC "."  .
  FullName     = ( initial | name ) { initial | name } .
  Title        = "Professor" | "Prof" | "Dr" | "Mr" | "Mrs" | "Ms" | "Mx" | "Miss" .
  Degree       =   "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci"
END Staff3.
```

but this has the unfortunate effect that it allows a name made of initials only, or a name to have an initial as its last component, as exemplified by

P. Terry D.

One might be tempted to be very rigid about punctuation, and insist that a surname incorporate a final period (if the person has no qualifictions) or a final comma (for persons that have qualifications. This is very restrictive, however.

Fortunately, it is easy to find a much better solution:

```
 PRODUCTIONS
   Staff4       = { Person } EOF .
   Person       = [ Title ] FullName { "," Degree } SYNC "."  .
   FullName     = NameLast { NameLast } .
   NameLast     = { initial } name .
   Title        = "Professor" | "Prof" | "Dr" | "Mr" | "Mrs" | "Ms" | "Mx" | "Miss" .
   Degree       =   "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci"
                  | "BSc(Hons)" | "BCom(Hons)" | "MA" | "MSc" | "PhD" .
 END Staff4.
```

Task 7 - Alternative description of EBNF

As hinted in the prac sheet, the trick here is to rework the productions that use meta braces for repetition by ones that use right recursion (to avoid LL(1) errors. Here is one possibility - and it does not use () parentheses either:

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

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  control  = CHR(0) .. CHR(31) .
  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        = Productions EOF .
  Productions = Production Productions | .
  Production  = nonterminal "=" Expression "." .
  Expression  = Term MoreTerms .
  MoreTerms   = "|" Term MoreTerms | .
  Term        = Factor MoreFactors .
  MoreFactors = Factor MoreFactors | .
  Factor      =   nonterminal
                | terminal
                | "[" Expression "]"
                | "(" Expression ")"
                | "{" Expression "}" .
END EBNF.
```

The above grammar matches the one given in the prac sheet. It does not, however, have the property of being able to describe itself any longer - we might argue that we need to be able to describe a nullable factor (the original could not do this). This might be achieved as follows:

```
...

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

but notice that this also allows one to write production rules like

```
A = b | c | | | .
```

which you might argue is a bit silly. Further reflection on this is left as a useful exercise!