Computer Science 3 - 2008

Programming Language Translation

Practical for Week 21, beginning 19 September 2008 - Solutions

Complete sources to these solutions can be found on the course WWW pages in the files `PRAC21A.ZIP` or `PRAC21AC.ZIP`

Task 2 - Extensions to the Simple Calculator

Exponentiation is a stronger operation than multiplication, so it has to be introduced deeper into the hierarchy (in fact this is discussed in the textbook, if you'd only thought of looking!) Functions like `sqrt` also take precedence over other operations:

```
COMPILER Calc  $CN
/* Simple four function calculator - extended
   P.D. Terry, Rhodes University, 2008 */

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

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .

IGNORE CHR(0) .. CHR(31)

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

Task 3 - Highland Gatherings

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 make sense to try

```
COMPILER Gath2 $CN
/* Describes the Pipe Band events at a Highland Gathering
   P.D. Terry, Rhodes University, 2008 */

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  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" .

END Gath2.
```

This is badly non LL(1) (It represents a continual wail of noise). We need to get some sort of break between bands at least:

```
COMPILER Gath3 $CN
/* Describes the Pipe Band events at a Highland Gathering
   P.D. Terry, Rhodes University, 2008 */

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS

  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"
                | "Strathspey" { "Strathspey" } "Reel" .

END Gath3.
```

Even this is non-LL(1). We can get an LL(1) grammar if we make the right kind of announcements:

```
COMPILER Gath4 $CN
/* Describes the Pipe Band events at a Highland Gathering
   P.D. Terry, Rhodes University, 2008 */

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS

  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" .

END Gath4.
```

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). So here is a much better solution:

```
COMPILER Gath5 $CN
/* Describes the Pipe Band events at a Highland Gathering
   P.D. Terry, Rhodes University, 2008 */

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 is non-LL(1) again, or of trying to write

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

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

Task 4 - 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:

```
COMPILER EBNF $CN
/* Parse a set of EBNF productions
   P.D. Terry, Rhodes University, 2008 */

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!

Task 5 - 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. Here is a suggested solution:

```
    COMPILER Parva $CN
    /* Parva level 1.5 grammar - Prac 21 extensions
       P.D. Terry, Rhodes University, 2008
       Grammar only */

    CHARACTERS
      lf           = CHR(10) .
      backslash    = CHR(92) .
      control      = CHR(0) .. CHR(31) .
      letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
      digit        = "0123456789" .
      nonzerodigit = "123456789" .
      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 most people thought of

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

*/

/* insisting that a number cannot have extraneous leading zeros */

      number     = "0" | nonzerodigit { 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 } "}" .

/* We need some more nonterminals for the new statement forms */

      Statement         =   Block | ConstDeclarations | VarDeclarations
                          | Assignment | IncOrDecStatement
                          | IfStatement | WhileStatement | RepeatStatement | ForStatement
                          | ReturnStatement | HaltStatement
                          | ReadStatement | WriteStatement | ";" .

/* Declarations remain the same as before */

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

/* To deal with statements like i = 6; i, j = 4, k; and i++; we need to manipulate the
   production for Assignment if we want to preserve an LL(1) grammar */

      Assignment        = Designator
                          (   { "," Designator } "=" Expression { "," Expression }
                            | "++"
                            | "--"
                          ) ";" .

/* Prefix increment/decrement statements like ++i; or ++list[7]; cause no LL(1) problems */

      IncOrDecStatement = ( "++" | "--" ) Designator ; .

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

```
     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 ] .

/* The Pascal-like "repeat" statement is almost trivial.  Note that we can make use of
   several Statements between "repeat" and "until"  (different in Java!) */

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

/* The for statement is straightforward, but introduces the concept of a list of
   expressions which is also used in the production for Expression itself */

     ForStatement      = "for" Designator "in" ExpList Statement .
     ExpList           = "(" Range { "," Range } ")" .
     Range             = Expression [ ".." Expression ] .

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

     WhileStatement    = "while" "(" Condition ")" Statement .
     ReturnStatement   = "return" ";" .
     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 | Constant
                         | "new" BasicType "[" Expression "]"
                         | "!" Factor | "(" Expression ")" .
     Type              = BasicType [ "[]" ] .
     BasicType         = "int" | "bool" .
     AddOp             = "+" | "-" | "||" .

/* The % operator has the same precedence as other multiplicative operators */

     MulOp             = "*" | "/" | "%" | "&&" .
     RelOp             = "==" | "!=" | "<" | "<=" | ">" | ">=" .
  END Parva.
```

Task 6 - Spoornet are looking for programmers

The wheels came off in many solutions. It is quite hard to get right, and one cannot easily find an LL(1) grammar that really matches the problem as set. Your situation was not helped by a rather poorly phrased question, which confused some groups. I apologise. Here is a simple first attempt, but with no safety regulations.

```
COMPILER Train1 $CN
/* Grammar for simple railway trains
   P.D. Terry, Rhodes University, 2008 */

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

PRODUCTIONS
  Train1         = { OneTrain } EOF .
  OneTrain       = LocoPart ( Passengers | FreightOrMixed ) "." .
  LocoPart       = "loco" { "loco" } .
  FreightOrMixed = Truck { Truck } ( "guard" | Passengers ) .
  Passengers     = { "coach" } "brake" .
  Truck          = "coal" | "open" | "cattle" | "fuel" | "cold" .
END Train1.
```

Here is an attempt at safety. But this one insists on at least two safe trucks in any train, and is not LL(1)

```
PRODUCTIONS
  Train2        = { OneTrain } EOF .
  OneTrain      = LocoPart ( Passengers | FreightOrMixed ) "." .
  LocoPart      = "loco" { "loco" } .
  FreightOrMixed = SafeTruck { AnyTruck } LastPart .
  LastPart      = "guard" | SafeTruck Passengers .
  Passengers    = { "coach" } "brake" .
  SafeTruck     = "coal" | "open" | "cattle" | "cold" .
  AnyTruck      = SafeTruck | "fuel" .
END Train2.
```

Why is it not LL(1)? We could apply all the theory of chapter 7, but maybe an example will suffice. Suppose we have a valid train like

```
loco coal coal coal coal coach brake
```

The first `coal` truck is parsed by the leading `SafeTruck` in `GoodsPart`. The next two `coal` trucks must be parsed by the repetitive part { `AnyTruck` }, but you can probably see that the last `coal` truck would have to be parsed by the alternative within `LastPart`. Unfortunately an LL(1) parser can't see far enough ahead to make that decision, and would be tempted to treat this last `coal` truck as part of the { `AnyTruck` } sequence.

Here is one that *is* LL(1)

```
PRODUCTIONS
  Train3        = { OneTrain } EOF .
  OneTrain      = LocoPart ( Passengers | FreightOrMixed ) "." .
  LocoPart      = "loco" { "loco" } .
  FreightOrMixed = SafeTruck MoreTrucks HumanPart .
  MoreTrucks    = { FuelTruck { FuelTruck } SafeTruck | SafeTruck } .
  HumanPart     = "guard" | Passengers .
  Passengers    = { "coach" } "brake" .
  SafeTruck     = "coal" | "open" | "cattle" | "cold" .
  FuelTruck     = "fuel" .
END Train3.
```

where you might notice that we have used a production rather like that used in describing the sequence of tunes in the Medley competition in Task 4. At first you might thing that this is, at last, a correct solution. But no, it isn't quite. This solution does not allow you to have a train like

```
loco loco open fuel fuel guard .
```

as the last fuel truck in a sequence has to be followed by at least one safe truck. The grammar does, however, allow trains like

```
loco open coach coach brake .
```

with only one truck in the goods section.

It is remarkable that something that at first sight looks so simple should turn out to be frustratingly difficult. Not being able to find an LL(1) grammar is not a train smash - one quite often cannot find an LL(1) grammar for a language, and for some years the LL(1) solution to this problem eluded us all. But it's usually worth a try, as parsers for LL(1) grammars are so easy to write. Here is a solution that seems to meet all requirements.

```
PRODUCTIONS
  Train4       = { OneTrain } EOF .
  OneTrain     = LocoPart [ SafeLoad | "guard" | Passengers ] "." .
  LocoPart     = "loco" { "loco" } .
  Passengers   = { "coach" } "brake" .
  SafeLoad     = SafeTruck RestSafeLoad .
  RestSafeLoad = SafeLoad | "guard" | Passengers | Fuel .
  Fuel         = FuelTruck { FuelTruck } ( SafeLoad | "guard" ) .
  SafeTruck    = "coal" | "open" | "cattle" | "cold" .
  FuelTruck    = "fuel" .
END Train4.
```

I was encouraged to see that many groups solved this problem, or got very close to doing so.