Computer Science 3 - 2013

Programming Language Translation

Practical for Week 21, beginning 16 September 2013 - 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

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, 2013 */

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 { "!" } .
  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 { "!" } .
  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, 2013 */

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 3 - 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, 2013 */

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 - 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 Parva1 $CN
/* Parva level 1.5 grammar  (Extended)
   This version uses C/Java/C#-like precedences for operators
   P.D. Terry, Rhodes University, 2013 */

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 something like
      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
  Parva1             = "void" identifier "(" ")" Block .
  Block              = "{" { Statement } "}" .

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

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

/* 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 ] .

/* AssignmentStatements require care to avoid LL(1) problems */

  AssignmentStatement = (   Designator ( AssignOp Expression | "++" | "--" )
                          | "++" Designator
                          | "--" Designator
                        ) ";" .

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

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

/* The if-then-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 the else clauses
   with the most recent if */

  IfStatement        = "if" "(" 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 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 DoWhileStatement must end with a semicolon!  */

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

/* The ForStatement needs to avoid using AssignmentStatement as one might be tempted to do.
   It is sensible to control a Modula-like for loop using a simple identifier rather than a general
   Designator for reasons that might be discussed later in the course */

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

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

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

  Type                = BasicType [ "[]" ] .
  BasicType           = "int" | "bool" .
  AddOp               = "+" | "-" | "||" .
  MulOp               = "*" | "/" | "%" | "&&" .
  RelOp               = "==" | "!=" | "<" | "<=" | ">" | ">=" .
  AssignOp            = "=" .

END Parva1.
```

Task 6 - Spoornet are looking for programmers

The problem suggested a Cocol grammar that describes correctly marshalled trains (`Trains.atg`):

```
COMPILER Trains $CN
/* Grammar for simple railway trains
   P.D. Terry, Rhodes University, 2013 */

IGNORECASE

COMMENTS FROM "(*" TO "*)" NESTED

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Trains    = { OneTrain } EOF .
  OneTrain  = LocoPart [ [ GoodsPart ] HumanPart ] SYNC "." .
  LocoPart  = "loco" { "loco" } .
  GoodsPart = Truck { Truck } .
  HumanPart = "brake" | { "coach" } "guard" .
  Truck     = "coal" | "closed" | "open" | "cattle" | "fuel" .
END Trains.
```

and went on to suggest modifying the grammar to build in restrictions that fuel trucks may not be marshalled immediately behind the locomotives, or immediately in front of a passenger coach.

In my experience the wheels come off in many attempts at solving this problem. It is quite hard to get right, and at first one may not easily find an LL(1) grammar that really matches the problem as set.

Given that passenger trains do not have a safety complication, one might be tempted to refactor the grammar to give the equivalent one below, which seems more closely to define a train in terms of the three ways in which it can be classified:

```
COMPILER Train1 $CN
/* Grammar for simple railway trains
   P.D. Terry, Rhodes University, 2013 */

IGNORECASE

COMMENTS FROM "(*" TO "*)" NESTED

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Train1         = { OneTrain } EOF .
  OneTrain       = LocoPart [ Passengers | FreightOrMixed ] SYNC "." .
  LocoPart       = "loco" { "loco" } .
  FreightOrMixed = Truck { Truck } ( "brake" | Passengers ) .
  Passengers     = { "coach" } "guard" .
  Truck          = "closed" | "coal" | "open" | "cattle" | "fuel" .
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 ] SYNC "." .
  LocoPart       = "loco" { "loco" } .
  FreightOrMixed = SafeTruck { AnyTruck } LastPart .
  LastPart       = "brake" | SafeTruck Passengers .
  Passengers     = { "coach" } "guard" .
  SafeTruck      = "closed" | "coal" | "open" | "cattle" .
  AnyTruck       = SafeTruck | "fuel" .
END Train2.
```

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

```
loco coal coal coal coal coach guard
```

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 ] SYNC "." .
  LocoPart       = "loco" { "loco" } .
  FreightOrMixed = SafeTruck MoreTrucks HumanPart .
  MoreTrucks     = { "fuel" { "fuel" } SafeTruck | SafeTruck } .
  HumanPart      = "brake" | Passengers .
  Passengers     = { "coach" } "guard" .
  SafeTruck      = "coal" | "closed" | "open" | "cattle" .
END Train3.
```

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 brake .
```

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

```
loco open coach coach guard .
```

with only one truck in the freight section.

It is remarkable that something that at first sight looks so simple might 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. But it's usually worth a try, as parsers for LL(1) grammars are so easy to write. The clue is to be found in a suggestion that one should factorize the grammar not to concentrate on the "obvious" types of train, but on the requirement that at any point along a train that might incorporate fuel trucks, the last part of the train should be "safe". Thus:

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

Every year, when I give this course, some students come up with better solutions that I had not thought of, and this year is no exception. I am not sure which of the five students who seem to have contributed to the solution below had the "Aha!" moment, but I was very impressed and hope that it really was a joint effort. This solution makes neat use of right recursion without using the {..} meta-brackets, a technique that is worth bearing in mind.

```
COMPILER Train5 $CN
/* Grammar for simple railway trains
   Original by P.D. Terry, Rhodes University, 2013
   Corrected, aided and abetted by Mikha Zeffert, Jessica Kent, Alisa Lochner,
   Kelvin Freese and Michael Abbott, 2013 class */

IGNORECASE

COMMENTS FROM "(*" TO "*)" NESTED

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Train5        = { OneTrain } EOF .
  OneTrain      = LocoPart [ GoodsPart | HumanPart | "brake" ] SYNC "." .
  LocoPart      = "loco" { "loco" } .
  HumanPart     = { "coach" } "guard" .
  GoodsPart     = SafeTruck [ HumanPart | GoodsPart | FuelPart | "brake" ] .
  FuelPart      = "fuel" [ GoodsPart | FuelPart | "brake" ] .
  SafeTruck     = "coal" | "closed" | "open" | "cattle" .
END Train5.
```

Terry Theorem One ("You can always tidy things up a bit more") might apply, and here is a slightly refactored version of the p

```
PRODUCTIONS
  Train6        = { OneTrain } EOF .
  OneTrain      = LocoPart [ GoodsPart | HumanPart | "brake" ] SYNC "." .
  LocoPart      = "loco" { "loco" } .
  HumanPart     = { "coach" } "guard" .
  GoodsPart     = SafeTruck [ HumanPart | SafePart ] .
  SafePart      = GoodsPart | FuelPart | "brake" .
  FuelPart      = "fuel" [ SafePart ] .
  SafeTruck     = "coal" | "closed" | "open" | "cattle" .
END Train6.
```