Computer Science 3 - 2009

Programming Language Translation

Practical for Week 21, beginning 21 September 2009 - 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

Extending the calculator grammar can be done in several ways. Here is a simple one of them, which corresponds to the approach taken 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, 2009 */

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

TOKENS
  decNumber  = 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 | [ "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 | [ "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!))). It is clearer like this than if one tries to simplify the definition of `Factor` still further to

```
    Factor     = ( "+" | "-" ) Primary { "!" } .
```

in which the interpretation of -4! would be (-4)! and not -(4!) as it should be.

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

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

PRODUCTIONS
  Calc5      = { Expression "=" } EOF .
  Expression = ["+" | "-" ] Term { "+" Term  | "-" Term } .
  Term       = Factor { "*" Factor | "/" Factor } .
  Factor     = Primary { "!" } | "abs" "(" Expression ")" .
  Primary    = decNumber | hexNumber | "(" Expression ")" .
END Calc5.
```

Several people suggested productions like this

```
Factor     = ( "+" | "-" ) Factor | Primary | "abs(" Expression ")" ) .
```

A terminal like `"abs("` is restrictive. It is usually better to allow white space to appears between method names and parameter lists if the user prefers this style.

Task 3 - Happy families

This was meant to be very straightforward and should have caused no difficulties. Here is one solution in the spirit of the exercise:

```
/* Describe a family
   P.D. Terry, Rhodes University, 2009
   Grammar only */

CHARACTERS
  control      = CHR(0) .. CHR(31) .
  uletter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lletter      = "abcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .

TOKENS
  name         = uletter { lletter | "'" uletter | "-" lletter } .
  number       = digit { digit } .

IGNORE control

PRODUCTIONS
  Family1      = { Section } .
  Section      = Surname | Parents | Grandparents | Children | Appendage .
  Surname      = "Family" ":" name { name } .
  Parents      = "Parents" ":" NameList .
  Grandparents = "Grandparents" ":" NameList .
  Children     = "Children" ":" NameList .
  NameList     = OnePerson { "," OnePerson } .
  OnePerson    = name { name } [ "(" "deceased" ")" ] .
  Appendage    = number (   "cat"   | "cats"   | "dog" | "dogs"
                          | [ "small" ] ( "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 } Surname { Generation } { Appendage } .
  Surname      = "Family" ":" name { name } .
  Generation   = ( "Parents" | "Grandparents" | "Children" ) ":" NameList .
  NameList     = OnePerson { "," OnePerson } .
  OnePerson    = name { name } [ "(" "deceased" ")" ] .
  Appendage    = number [ "small" ] ( "cat"   | "cats"   | "dog" | "dogs" |
                                      "house" | "houses" | "car" | "cars" ) .
END Family2.
```

Three more 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) we could have used the terminal `name` instead of listing the specific possessions in the family.

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

```
item = lletter { lletter } .
```

and changed the production

```
Appendage    = number item { item } .
```

Task 4 - One for the Musicians in our Midst (but the rest of you should do it too)

This is straightforward, but note the way in which an `eol` 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 solfa key words as ordinary words - for example "so". 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 sequsnces like `Tom!!,Dick,Harry` as making up one word.

```
COMPILER Solfa $CN
/* Describe the words and notes of a tune using tonic solfa
   P.D. Terry, Rhodes University, 2009
   Grammar only */

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

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

IGNORE control - eol

PRODUCTIONS
  Solfa      = { Line } .
  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.
```

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 grammar  - Coco/R for C# (EBNF)
   P.D. Terry, Rhodes University, 2009
   Extended for prac 21
   Grammar only */

CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  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 most people thought 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 numbers to be of the various forms suggested is easy enough */

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

  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 | AssignmentStatement
                       | IfStatement | WhileStatement | ReturnStatement | HaltStatement
                       | ReadStatement | WriteStatement
                       | ForStatement | DoWhileStatement
                       | BreakStatement | ContinueStatement | ";" .

/* 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 ] .

/* Factoring out Assignment from AssignmentStatement makes for ease in defining the ForStatement */

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

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

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

/* The extension to the IfStatement is easy, though it leads to a non-critical LL(1) warning  */

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

/* Remember that the DoWhileStatement and GoToStatement end with a semicolon!  */

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

/* The ForStatement needs to avoid using "AssignmentStatement" as many people tried to do  */

  ForStatement       = "for" "("
                           [ [ BasicType ] identifier "=" Expression ";" ]
                           [ Condition ] ";"
                           [ Assignment ]
                         ")" 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" ";" .

/* 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 .
  Expression         = AddExp [ RelOp AddExp ] .
  AddExp             = [ "+" | "-" ] Term { AddOp Term } .
  Term               = Factor { MulOp Factor } .
  Factor             =   Designator | Constant
                       | "new" BasicType "[" Expression "]"

/* Type conversion functions are easy to add syntactically
   We are not using the (type) casting syntax as found in the C family.
   A function should be notated as a function */

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

  Type               = BasicType [ "[]" ] .

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

  BasicType          = "int" | "bool" | "char" .
  AddOp              = "+" | "-" | "||" .
```

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

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

Task 6 - XML

The grammar here is quite simple - perhaps the only tricky bit is to get the token definitions correct. Note that the context free grammar here is far too permissive - it cannot check the spellings of the various tags (at least, not at this stage), and of course the spelling is crucially important.

```
COMPILER XML $CN
/* Parse a set of simple XML elements (no attributes)
   P.D. Terry, Rhodes University, 2009 */

CHARACTERS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline   = "_" .
  intag     = letter + "0123456789_.-" .
  inword    = ANY - "<" .
  incomment = ANY - "-" .

TOKENS
  opentag  = "<"  ( letter | lowline ) { intag } ">" .
  emptytag = "<"  ( letter | lowline ) { intag } "/>" .
  closetag = "</" ( letter | lowline ) { intag } ">" .
  word     = inword { inword } .

PRAGMAS
  comment = "<!--" { incomment | '-' incomment } "-->" .

IGNORE  CHR(0) .. CHR(31)

PRODUCTIONS
   XML       = Element .

   Element    =
                opentag
                {   Element
                  | word
                  | emptytag
                }
                closetag .

END XML.
```

Pragmas, like comments can appear anywhere. We have not covered pragmas yet, and I apologise for not noticing this when I set the question.