

# Computer Science 3 - 2010

## Programming Language Translation

### Practical for Week 21, beginning 13 September 2010 - 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 106 of the textbook, and you were invited to extend it to allow for exponentiation, parentheses, leading unary + or - operators, a `sqrt()` function, numbers with decimal points and so on.

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.

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, 2010 */

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 ")"
              | "sqrt" "(" 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 ")"
              | "sqrt" "(" 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 [ "^" Factor ]
              | "sqrt" "(" Expression ")" .
  Primary   = decNumber | hexNumber | "(" Expression ")" .

END Calc3.
```

```

PRODUCTIONS
  Calc4      = { Expression "=" } EOF .
  Expression = Term { "+" Term | "-" Term } .
  Term       = Factor { "*" Factor | "/" Factor } .
  Factor     = ( "+" | "-" ) Factor | Primary [ "^" Factor ]
              | "sqrt" "(" Expression ")" .
  Primary    = decNumber | hexNumber | "(" Expression )" .
END Calc4.

```

It may be tempting to suggest a production like this

```

Primary    = ( decNumber | hexNumber | "(" Expression )"
              | "sqrt(" Expression )"
            ) .

```

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

### Task 3 - Meet the staff in my department

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 academic staff in a department
   Non-LL(1), and will not work properly
   P.D. Terry, Rhodes University, 2010 */

CHARACTERS
  uLetter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter = "abcdefghijklmnopqrstuvwxyz" .
  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" | "Dr" | "Mr" | "Mrs" | "Ms" | "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" | "Dr" | "Mr" | "Mrs" | "Ms" | "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" | "Dr" | "Mr" | "Mrs" | "Ms" | "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 qualifications) 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" | "Dr" | "Mr" | "Mrs" | "Ms" | "Miss" .
Degree      = "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci"
              | "BSc(Hons)" | "BCom(Hons)" | "MA" | "MSc" | "PhD" .
END Staff4.

```

#### 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. Here is a 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, 2010 */

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

TOKENS

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

/* Restricting the form of numbers to forbid leading zeros is easy enough */
number     = "0" | nonZeroDigit { digit } .

/* 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
  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
                    | ForStatement BreakStatement
                    | ContinueStatement RepeatStatement
                    | ";"
                  ) .

/* 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 Pascal-like "repeat" statement is almost trivial. Note that we can use
   "repeat" { Statement } "until" ...
   rather than being restricted to
   "do" Statement "while" ...
   as in the c languages. Why is this so? */

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

/* The ForStatement needs to avoid using AssignmentStatement as one might be tempted to do */

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" ";" .

/* 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 .

/* To handle Expressions with Java-inspired precedence we might proceed as follows. Note the
   use of the { } metabrackets in all but one of the following productions.
   Type conversion functions are easy to add syntactically.
   We are not using the (type) casting syntax as found in the c family.

```

```

    (It is left as an exercise to define the grammar so that you can do so.)
    I think a function (or pseudo-function) should be notated as a function! */

Expression      = AndExp { "|" AndExp } .
AndExp          = EqLExp { "&&" EqLExp } .
EqLExp          = RelExp { EqLOp RelExp } .
RelExp          = AddExp [ RelOp AddExp
                        | "in" ExpList
                        ] . // This must use [ ] not { } Do you see why?
AddExp          = MulExp { AddOp MulExp } .
MulExp         = Factor { MulOp Factor } .
Factor          = Primary | ( "+" | "-" | "!" ) Factor .
Primary         = Designator | Constant
                | "new" BasicType [ " Expression "]"
                | [ "char" | "int" ] "( Expression )" .
ExpList         = "<" Range { "," Range } ">" .
Range           = Expression [ ".." Expression ] .

Type            = BasicType [ "[" ] .

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

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

/* We need to classify the various operators differently */

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

END Parva1.

```

## Task 5 - When all else fails, look up what you need in the index

The description of a book index has always produced some innovative and imaginative solutions whenever I have used it as an example. There is no simple correct answer - looking at the example given usually leads to students deriving a set of productions to which one can respond "but what if you had an entry in the index reading like this" and finding another plausible one. Here is one suggested solution, in which I have played tricks with the selection of character sets. An important idea is to factorize the solution to reflect the important ideas that the entries in an index have two main components - a "subject" and a "list of references".

It may be tempting to try to define a "space token". This is futile; spaces between tokens are always ignored by scanners produced by Coco. One *can* define tokens that *contain* spaces, but this is only of much use in tokens like strings, which are demarcated by matching quotes or brackets.

```

COMPILER Index1 $CN
/* Grammar describing index in a book
   P.D. Terry, Rhodes University, 2010 */

CHARACTERS
/* Notice the careful and unusual choice of character sets */
control      = CHR(0) .. CHR(31) .
digit        = "0123456789" .
startword    = "ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz(' + '" .
inword       = startword + digit + "+)" .
lf           = CHR(10) .

TOKENS
/* Notice the careful and unusual definition for word */
word         = startword { inword } .
number       = digit { digit } .
EOL          = lf .

IGNORE control - lf

```

```

PRODUCTIONS
Index1  = { Entry } EOF .
Entry   = Key References SYNC EOL .
Key     = word { "," word | word } .
References = DirectRefs | CrossRef .
DirectRefs = PageRefs { "," PageRefs } .
PageRefs  = number [ "-" number ] | Appendix .
Appendix  = "Appendix" number .
CrossRef  = "---" "see" Key .
END Index1.

```

One year when I set this exercise, I received an intriguing submission on the following lines (this has been modified very slightly from the original submission). Compare it with the one above and try to work out whether this is better, or whether it is incorrect.

```

COMPILER Index2 $CN
/* Grammar describing index in a book
   Based on code by Wright, Nottingham, Ngwaile and Charlton
   This version by P.D. Terry, Rhodes University, 2007 */

CHARACTERS
control = CHR(0) .. CHR(31) .
digit   = "0123456789" .
inword  = ANY - control - digit - ",- " .
lf      = CHR(10) .
cr      = CHR(13) .

TOKENS
word    = inword { inword | digit | "-" } .
number  = digit { digit } .
EOL     = cr [ lf ] | lf .

IGNORE control - cr - lf

PRODUCTIONS
Index2  = { Entry } EOF .
Entry   = Key References SYNC EOL .
Key     = Words { "," Words } .
Words   = word { word } .
References = DirectRefs | CrossRef .
DirectRefs = PageRefs { "," PageRefs } .
PageRefs  = number [ "-" number ] | Appendix .
Appendix  = "Appendix" number .
CrossRef  = "---" "see" Key .
END Index2.

```